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

Volume 2 (2A, 2B, 2C, & 2D): Instruction Set Reference, A-Z

NOTE: The Intel 64 and IA-32 Architectures Software Developer's Manual consists of four volumes: *Basic Architecture*, Order Number 253665; *Instruction Set Reference, A-Z*, Order Number 325383; *System Programming Guide*, Order Number 325384; *Model-Specific Registers,* Order Number 335592. Refer to all four volumes when evaluating your design needs.

Order Number: 325383-093US
September 2026

Notices & Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

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

All product plans and roadmaps are subject to change without notice.

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

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Code names are used by Intel to identify products, technologies, or services that are in development and not publicly available. These are not “commercial” names and not intended to function as trademarks.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document, with the sole exception that a) you may publish an unmodified copy and b) code, identified as Sample Code in this document is licensed subject to the Zero-Clause BSD open source license (0BSD), https://opensource.org/licenses/0BSD. You may create software implementations based on this document and in compliance with the foregoing that are intended to execute on the Intel product(s) referenced in this document. No rights are granted to create modifications or derivatives of this document.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

CONTENTS

PAGE

CHAPTER 1
ABOUT THIS MANUAL
1.1 OVERVIEW OF VOLUME 2A, 2B, 2C, AND 2D: INSTRUCTION SET REFERENCE 1-1

CHAPTER 2
INSTRUCTION FORMAT
2.1 INSTRUCTION FORMAT FOR PROTECTED MODE, REAL-ADDRESS MODE, AND VIRTUAL-8086 MODE 2-1
2.1.1 Instruction Prefixes 2-1
2.1.2 Opcodes 2-2
2.1.3 ModR/M and SIB Bytes 2-3
2.1.4 Displacement and Immediate Bytes 2-3
2.1.5 Addressing-Mode Encoding of ModR/M and SIB Bytes 2-3
2.2 IA-32E MODE 2-7
2.2.1 REX Prefixes 2-7
2.2.1.1 Encoding 2-8
2.2.1.2 More on REX Prefix Fields 2-8
2.2.1.3 Displacement 2-11
2.2.1.4 Direct Memory-Offset MOVs 2-11
2.2.1.5 Immediates 2-11
2.2.1.6 RIP-Relative Addressing 2-11
2.2.1.7 Default 64-Bit Operand Size 2-12
2.2.2 Additional Encodings for Control and Debug Registers 2-12
2.3 INTEL® ADVANCED VECTOR EXTENSIONS (INTEL® AVX) 2-12
2.3.1 Instruction Format 2-12
2.3.2 VEX and the LOCK prefix 2-13
2.3.3 VEX and the 66H, F2H, and F3H prefixes 2-13
2.3.4 VEX and the REX prefix 2-13
2.3.5 The VEX Prefix 2-13
2.3.5.1 VEX Byte 0, bits[7:0] 2-15
2.3.5.2 VEX Byte 1, bit [7] - 'R' 2-15
2.3.5.3 3-byte VEX byte 1, bit[6] - 'X' 2-16
2.3.5.4 3-byte VEX byte 1, bit[5] - 'B' 2-16
2.3.5.5 3-byte VEX byte 2, bit[7] - 'W' 2-16
2.3.5.6 2-byte VEX Byte 1, bits[6:3] and 3-byte VEX Byte 2, bits [6:3]- 'vvvv' the Source or Dest Register Specifier 2-16
2.3.6 Instruction Operand Encoding and VEX.vvvv, ModR/M 2-17
2.3.6.1 3-byte VEX byte 1, bits[4:0] - "m-mmmm" 2-18
2.3.6.2 2-byte VEX byte 1, bit[2], and 3-byte VEX byte 2, bit [2]- "L" 2-18
2.3.6.3 2-byte VEX byte 1, bits[1:0], and 3-byte VEX byte 2, bits [1:0]- "pp" 2-19
2.3.7 The Opcode Byte 2-19
2.3.8 The ModR/M, SIB, and Displacement Bytes 2-19
2.3.9 The Third Source Operand (Immediate Byte) 2-19
2.3.10 Intel® AVX Instructions and the Upper 128-bits of YMM registers 2-19
2.3.10.1 Vector Length Transition and Programming Considerations 2-19
2.3.11 Intel® AVX Instruction Length 2-20
2.3.12 Vector SIB (VSIB) Memory Addressing 2-20
2.3.12.1 64-bit Mode VSIB Memory Addressing 2-21
2.4 INTEL® ADVANCED MATRIX EXTENSIONS (INTEL® AMX) 2-21
2.5 INTEL® AVX AND INTEL® SSE INSTRUCTION EXCEPTION CLASSIFICATION 2-22
2.5.1 Exceptions Type 1 (Aligned Memory Reference) 2-26
2.5.2 Exceptions Type 2 (>=16 Byte Memory Reference, Unaligned) 2-27
2.5.3 Exceptions Type 3 (<16 Byte Memory Argument) 2-28
2.5.4 Exceptions Type 4 (>=16 Byte Mem Arg, No Alignment, No Floating-point Exceptions) 2-29
2.5.5 Exceptions Type 5 (<16 Byte Mem Arg and No FP Exceptions) 2-30
2.5.6 Exceptions Type 6 (VEX-Encoded Instructions without Legacy SSE Analogues) 2-31
2.5.7 Exceptions Type 7 (No FP Exceptions, No Memory Arg) 2-32
2.5.8 Exceptions Type 8 (AVX and No Memory Argument) 2-32
2.5.9 Exceptions Type 11 (VEX-only, Mem Arg, No AC, Floating-point Exceptions) 2-33
2.5.10 Exceptions Type 12 (VEX-only, VSIB Mem Arg, No AC, No Floating-point Exceptions) 2-34
2.6 VEX ENCODING SUPPORT FOR GPR INSTRUCTIONS 2-34
2.6.1 Exceptions Type 13 (VEX-Encoded GPR Instructions) 2-35

PAGE

2.6.2 Exceptions Type 14 (CMPCCXADD) ... 2-35
2.7 INTEL® AVX-512 ENCODING ... 2-36
2.7.1 Instruction Format and EVEX ... 2-36
2.7.2 Register Specifier Encoding and EVEX ... 2-39
2.7.3 Opmask Register Encoding ... 2-40
2.7.4 Masking Support in EVEX ... 2-40
2.7.5 Compressed Displacement (disp8*N) Support in EVEX ... 2-41
2.7.6 EVEX Encoding of Broadcast/Rounding/SAE Support ... 2-42
2.7.7 Embedded Broadcast Support in EVEX ... 2-42
2.7.8 Static Rounding Support in EVEX ... 2-42
2.7.9 SAE Support in EVEX ... 2-42
2.7.10 Vector Length Orthogonality ... 2-42
2.7.11 #UD Equations for EVEX ... 2-43
2.7.11.1 State Dependent #UD ... 2-43
2.7.11.2 Opcode Independent #UD ... 2-43
2.7.11.3 Opcode Dependent #UD ... 2-44
2.7.12 Device Not Available ... 2-45
2.7.13 Scalar Instructions ... 2-45
2.8 EXCEPTION CLASSIFICATIONS OF EVEX-ENCODED INSTRUCTIONS ... 2-45
2.8.1 Exceptions Type E1 and E1NF of EVEX-Encoded Instructions ... 2-48
2.8.2 Exceptions Type E2 of EVEX-Encoded Instructions ... 2-50
2.8.3 Exceptions Type E3 and E3NF of EVEX-Encoded Instructions ... 2-51
2.8.4 Exceptions Type E4 and E4NF of EVEX-Encoded Instructions ... 2-53
2.8.5 Exceptions Type E5 and E5NF ... 2-55
2.8.6 Exceptions Type E6 and E6NF ... 2-58
2.8.7 Exceptions Type E7NM ... 2-59
2.8.8 Exceptions Type E9 and E9NF ... 2-60
2.8.9 Exceptions Type E10 and E10NF ... 2-62
2.8.10 Exceptions Type E11 (EVEX-only, Mem Arg, No AC, Floating-point Exceptions) ... 2-64
2.8.11 Exceptions Type E12 and E12NP (VSIB Mem Arg, No AC, No Floating-point Exceptions) ... 2-66
2.9 EXCEPTION CLASSIFICATIONS OF OPMASK INSTRUCTIONS, TYPE K20 AND TYPE K21 ... 2-67
2.9.1 Exceptions Type K20 ... 2-67
2.9.2 Exceptions Type K21 ... 2-68
2.10 INTEL® AMX INSTRUCTION EXCEPTION CLASSES ... 2-69

CHAPTER 3
INSTRUCTION SET REFERENCE, A-L
3.1 INTERPRETING THE INSTRUCTION REFERENCE PAGES ... 3-1
3.1.1 Instruction Format ... 3-1
3.1.1.1 Opcode Column in the Instruction Summary Table (Instructions without VEX Prefix) ... 3-2
3.1.1.2 Opcode Column in the Instruction Summary Table (Instructions with VEX prefix) ... 3-3
3.1.1.3 Instruction Column in the Opcode Summary Table ... 3-5
3.1.1.4 Operand Encoding Column in the Instruction Summary Table ... 3-8
3.1.1.5 64/32-bit Mode Column in the Instruction Summary Table ... 3-8
3.1.1.6 CPUID Support Column in the Instruction Summary Table ... 3-9
3.1.1.7 Description Column in the Instruction Summary Table ... 3-9
3.1.1.8 Description Section ... 3-9
3.1.1.9 Operation Section ... 3-9
3.1.1.10 Intel® C/C++ Compiler Intrinsics Equivalents Section ... 3-12
3.1.1.11 Flags Affected Section ... 3-14
3.1.1.12 FPU Flags Affected Section ... 3-15
3.1.1.13 Protected Mode Exceptions Section ... 3-15
3.1.1.14 Real-Address Mode Exceptions Section ... 3-16
3.1.1.15 Virtual-8086 Mode Exceptions Section ... 3-16
3.1.1.16 Floating-Point Exceptions Section ... 3-16
3.1.1.17 SIMD Floating-Point Exceptions Section ... 3-16
3.1.1.18 Compatibility Mode Exceptions Section ... 3-17
3.1.1.19 64-Bit Mode Exceptions Section ... 3-17
3.2 INTEL® AMX CONSIDERATIONS ... 3-17
3.2.1 Implementation Parameters ... 3-17
3.2.2 Helper Functions ... 3-17
3.3 INSTRUCTIONS (A-L) ... 3-18
AAA—ASCII Adjust After Addition ... 3-1
AAD—ASCII Adjust AX Before Division ... 3-3

PAGE

AAM—ASCII Adjust AX After Multiply 3-5
AAS—ASCII Adjust AL After Subtraction 3-7
ADC—Add With Carry 3-9
ADCX—Unsigned Integer Addition of Two Operands With Carry Flag 3-12
ADD—Add 3-14
ADDPD—Add Packed Double Precision Floating-Point Values 3-16
ADDPS—Add Packed Single Precision Floating-Point Values 3-19
ADDSD—Add Scalar Double Precision Floating-Point Values 3-22
ADDSS—Add Scalar Single Precision Floating-Point Values 3-24
ADDSUBPD—Packed Double Precision Floating-Point Add/Subtract 3-26
ADDSUBPS—Packed Single Precision Floating-Point Add/Subtract 3-28
ADOX — Unsigned Integer Addition of Two Operands With Overflow Flag 3-31
AESDEC—Perform One Round of an AES Decryption Flow 3-33
AESDEC128KL—Perform Ten Rounds of AES Decryption Flow With Key Locker Using 128-Bit Key 3-35
AESDEC256KL—Perform 14 Rounds of AES Decryption Flow With Key Locker Using 256-Bit Key 3-37
AESDECLAST—Perform Last Round of an AES Decryption Flow 3-39
AESDECWIDE128KL—Perform Ten Rounds of AES Decryption Flow With Key Locker on 8 Blocks Using 128-Bit Key 3-41
AESDECWIDE256KL—Perform 14 Rounds of AES Decryption Flow With Key Locker on 8 Blocks Using 256-Bit Key . 3-43
AESENC—Perform One Round of an AES Encryption Flow 3-45
AESENC128KL—Perform Ten Rounds of AES Encryption Flow With Key Locker Using 128-Bit Key 3-47
AESENC256KL—Perform 14 Rounds of AES Encryption Flow With Key Locker Using 256-Bit Key 3-49
AESENCLAST—Perform Last Round of an AES Encryption Flow 3-51
AESENCWIDE128KL—Perform Ten Rounds of AES Encryption Flow With Key Locker on 8 Blocks Using 128-Bit Key 3-53
AESENCWIDE256KL—Perform 14 Rounds of AES Encryption Flow With Key Locker on 8 Blocks Using 256-Bit Key . 3-55
AESIMC—Perform the AES InvMixColumn Transformation 3-57
AESKEYGENASSIST—AES Round Key Generation Assist 3-58
AND—Logical AND 3-60
ANDN—Logical AND NOT 3-62
ANDNPD—Bitwise Logical AND NOT of Packed Double Precision Floating-Point Values 3-63
ANDNPS—Bitwise Logical AND NOT of Packed Single Precision Floating-Point Values 3-66
ANDPD—Bitwise Logical AND of Packed Double Precision Floating-Point Values 3-69
ANDPS—Bitwise Logical AND of Packed Single Precision Floating-Point Values 3-72
ARPL—Adjust RPL Field of Segment Selector 3-75
BEXTR—Bit Field Extract 3-77
BLENDPD—Blend Packed Double Precision Floating-Point Values 3-78
BLENDPS—Blend Packed Single Precision Floating-Point Values 3-80
BLENDVPD—Variable Blend Packed Double Precision Floating-Point Values 3-82
BLENDVPS—Variable Blend Packed Single Precision Floating-Point Values 3-84
BLSI—Extract Lowest Set Isolated Bit 3-87
BLSMSK—Get Mask Up to Lowest Set Bit 3-88
BLSR—Reset Lowest Set Bit 3-89
BNDCL—Check Lower Bound 3-90
BNDCU/BNDCN—Check Upper Bound 3-92
BNDLDX—Load Extended Bounds Using Address Translation 3-94
BNDMK—Make Bounds 3-97
BNDMOV—Move Bounds 3-99
BNDSTX—Store Extended Bounds Using Address Translation 3-102
BOUND—Check Array Index Against Bounds 3-105
BSF—Bit Scan Forward 3-107
BSR—Bit Scan Reverse 3-109
BSWAP—Byte Swap 3-111
BT—Bit Test 3-112
BTC—Bit Test and Complement 3-114
BTR—Bit Test and Reset 3-116
BTS—Bit Test and Set 3-118
BZHI—Zero High Bits Starting with Specified Bit Position 3-120
CALL—Call Procedure 3-121
CBW/CWDE/CDQE—Convert Byte to Word/Convert Word to Doubleword/Convert Doubleword to Quadword 3-138
CLAC—Clear AC Flag in EFLAGS Register 3-139

PAGE

CLC—Clear Carry Flag 3-140
CLD—Clear Direction Flag 3-141
CLDEMOTE—Cache Line Demote 3-142
CLFLUSH—Flush Cache Line 3-144
CLFLUSHOPT—Flush Cache Line Optimized 3-146
CLI—Clear Interrupt Flag 3-148
CLRSSBSY—Clear Busy Flag in a Supervisor Shadow Stack Token 3-150
CLTS—Clear Task-Switched Flag in CR0 3-152
CLUI—Clear User Interrupt Flag 3-153
CLWB—Cache Line Write Back 3-154
CMC—Complement Carry Flag 3-156
CMOVcc—Conditional Move 3-157
CMP—Compare Two Operands 3-161
CMPccXADD—Compare and Add if Condition is Met 3-163
CMPPD—Compare Packed Double Precision Floating-Point Values 3-168
CMPPS—Compare Packed Single Precision Floating-Point Values 3-175
CMPS/CMPSB/CMPSW/CMPSD/CMPSQ—Compare String Operands 3-181
CMPSD—Compare Scalar Double Precision Floating-Point Value 3-185
CMPSS—Compare Scalar Single Precision Floating-Point Value 3-189
CMPXCHG—Compare and Exchange 3-194
CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes 3-196
COMISD—Compare Scalar Ordered Double Precision Floating-Point Values and Set EFLAGS 3-199
COMISS—Compare Scalar Ordered Single Precision Floating-Point Values and Set EFLAGS 3-201
CPUID—CPU Identification 3-203
CRC32—Accumulate CRC32 Value 3-205
CVTDQ2PD—Convert Packed Doubleword Integers to Packed Double Precision Floating-Point Values 3-208
CVTDQ2PS—Convert Packed Doubleword Integers to Packed Single Precision Floating-Point Values 3-211
CVTPD2DQ—Convert Packed Double Precision Floating-Point Values to Packed Doubleword Integers 3-214
CVTPD2PI—Convert Packed Double Precision Floating-Point Values to Packed Dword Integers 3-218
CVTPD2PS—Convert Packed Double Precision Floating-Point Values to Packed Single Precision Floating-Point Values 3-219
CVTPI2PD—Convert Packed Dword Integers to Packed Double Precision Floating-Point Values 3-223
CVTPI2PS—Convert Packed Dword Integers to Packed Single Precision Floating-Point Values 3-224
CVTPS2DQ—Convert Packed Single Precision Floating-Point Values to Packed Signed Doubleword Integer Values 3-225
CVTPS2PD—Convert Packed Single Precision Floating-Point Values to Packed Double Precision Floating-Point Values 3-228
CVTPS2PI—Convert Packed Single Precision Floating-Point Values to Packed Dword Integers 3-231
CVTSD2SI—Convert Scalar Double Precision Floating-Point Value to Signed Integer 3-232
CVTSD2SS—Convert Scalar Double Precision Floating-Point Value to Scalar Single Precision Floating-Point Value 3-234
CVTSI2SD—Convert Signed Integer to Scalar Double Precision Floating-Point Value 3-236
CVTSI2SS—Convert Signed Integer to Scalar Single Precision Floating-Point Value 3-238
CVTSS2SD—Convert Scalar Single Precision Floating-Point Value to Scalar Double Precision Floating-Point Value 3-240
CVTSS2SI—Convert Scalar Single Precision Floating-Point Value to Signed Integer 3-242
CVTTPD2DQ—Convert with Truncation Packed Double Precision Floating-Point Values to Packed Doubleword Integers 3-244
CVTTPD2PI—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Dword Integers 3-248
CVTTPS2DQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Signed Doubleword Integer Values 3-249
CVTTPS2PI—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Dword Integers 3-252
CVTTSD2SI—Convert With Truncation Scalar Double Precision Floating-Point Value to Signed Integer 3-253
CVTTSS2SI—Convert With Truncation Scalar Single Precision Floating-Point Value to Signed Integer 3-255
CWD/CDQ/CQO—Convert Word to Doubleword/Convert Doubleword to Quadword 3-257
DAA—Decimal Adjust AL After Addition 3-258
DAS—Decimal Adjust AL After Subtraction 3-260
DEC—Decrement by 1 3-262
DIV—Unsigned Divide 3-264
DIVPD—Divide Packed Double Precision Floating-Point Values 3-267
DIVPS—Divide Packed Single Precision Floating-Point Values 3-270
DIVSD—Divide Scalar Double Precision Floating-Point Value 3-273

PAGE

DIVSS—Divide Scalar Single Precision Floating-Point Values . . . 3-275
DPPD—Dot Product of Packed Double Precision Floating-Point Values . . . 3-277
DPPS—Dot Product of Packed Single Precision Floating-Point Values . . . 3-279
EMMS—Empty MMX Technology State . . . 3-282
ENCODEKEY128—Encode 128-Bit Key With Key Locker . . . 3-283
ENCODEKEY256—Encode 256-Bit Key With Key Locker . . . 3-285
ENDBR32—Terminate an Indirect Branch in 32-bit and Compatibility Mode . . . 3-287
ENDBR64—Terminate an Indirect Branch in 64-bit Mode . . . 3-288
ENQCMD—Enqueue Command . . . 3-289
ENQCMDS—Enqueue Command Supervisor . . . 3-292
ENTER—Make Stack Frame for Procedure Parameters . . . 3-295
ERETS—Event Return to Supervisor . . . 3-298
ERETU—Event Return to User . . . 3-301
EXTRACTPS—Extract Packed Floating-Point Values . . . 3-305
F2XM1—Compute 2x–1 . . . 3-307
FABS—Absolute Value . . . 3-309
FADD/FADDP/FIADD—Add . . . 3-311
FBLD—Load Binary Coded Decimal . . . 3-314
FBSTP—Store BCD Integer and Pop . . . 3-316
FCHS—Change Sign . . . 3-318
FCLEX/FNCLEX—Clear Exceptions . . . 3-320
FCMOVcc—Floating-Point Conditional Move . . . 3-322
FCOM/FCOMP/FCOMPP—Compare Floating-Point Values . . . 3-324
FCOMI/FCOMIP/ FUCOMI/FUCOMIP—Compare Floating-Point Values and Set EFLAGS . . . 3-327
FCOS—Cosine . . . 3-330
FDECSTP—Decrement Stack-Top Pointer . . . 3-332
FDIV/FDIVP/FIDIV—Divide . . . 3-333
FDIVR/FDIVRP/FIDIVR—Reverse Divide . . . 3-336
FFREE—Free Floating-Point Register . . . 3-339
FICOM/FICOMP—Compare Integer . . . 3-340
FILD—Load Integer . . . 3-342
FINCSTP—Increment Stack-Top Pointer . . . 3-344
FINIT/FNINIT—Initialize Floating-Point Unit . . . 3-345
FIST/FISTP—Store Integer . . . 3-347
FISTTP—Store Integer With Truncation . . . 3-350
FLD—Load Floating-Point Value . . . 3-352
FLD1/FLDL2T/FLDL2E/FLDPI/FLDLG2/FLDLN2/FLDZ—Load Constant . . . 3-354
FLDCW—Load x87 FPU Control Word . . . 3-356
FLDENV—Load x87 FPU Environment . . . 3-358
FMUL/FMULP/FIMUL—Multiply . . . 3-360
FNOP—No Operation . . . 3-363
FPATAN—Partial Arctangent . . . 3-364
FPREM—Partial Remainder . . . 3-366
FPREM1—Partial Remainder . . . 3-368
FPTAN—Partial Tangent . . . 3-370
FRNDINT—Round to Integer . . . 3-372
FRSTOR—Restore x87 FPU State . . . 3-373
FSAVE/FNSAVE—Store x87 FPU State . . . 3-375
FSCALE—Scale . . . 3-378
FSIN—Sine . . . 3-380
FSINCOS—Sine and Cosine . . . 3-382
FSQRT—Square Root . . . 3-384
FST/FSTP—Store Floating-Point Value . . . 3-386
FSTCW/FNSTCW—Store x87 FPU Control Word . . . 3-388
FSTENV/FNSTENV—Store x87 FPU Environment . . . 3-390
FSTSW/FNSTSW—Store x87 FPU Status Word . . . 3-392
FSUB/FSUBP/FISUB—Subtract . . . 3-394
FSUBR/FSUBRP/FISUBR—Reverse Subtract . . . 3-397
FTST—TEST . . . 3-400

PAGE

FUCOM/FUCOMP/FUCOMPP—Unordered Compare Floating-Point Values 3-402
FXAM—Examine Floating-Point 3-405
FXCH—Exchange Register Contents 3-407
FXRSTOR—Restore x87 FPU, MMX, XMM, and MXCSR State 3-409
FXSAVE—Save x87 FPU, MMX Technology, and SSE State 3-412
FXTRACT—Extract Exponent and Significand 3-420
FYL2X—Compute y * log2x 3-422
FYL2XP1—Compute y * log2(x +1) 3-424
GF2P8AFFINEINVQB—Galois Field Affine Transformation Inverse 3-426
GF2P8AFFINEQB—Galois Field Affine Transformation 3-429
GF2P8MULB—Galois Field Multiply Bytes 3-431
HADDPD—Packed Double Precision Floating-Point Horizontal Add 3-433
HADDPS—Packed Single Precision Floating-Point Horizontal Add 3-436
HLT—Halt 3-439
HRESET—History Reset 3-440
HSUBPD—Packed Double Precision Floating-Point Horizontal Subtract 3-442
HSUBPS—Packed Single Precision Floating-Point Horizontal Subtract 3-445
IDIV—Signed Divide 3-448
IMUL—Signed Multiply 3-451
IN—Input From Port 3-455
INC—Increment by 1 3-457
INCSSPD/INCSSPQ—Increment Shadow Stack Pointer 3-459
INSERTPS—Insert Scalar Single Precision Floating-Point Value 3-461
INS/INSB/INSW/INSD—Input from Port to String 3-464
INT n/INTO/INT3/INT1—Call to Interrupt Procedure 3-467
INVD—Invalidate Internal Caches 3-483
INVLPG—Invalidate TLB Entries 3-485
INVPCID—Invalidate Process-Context Identifier 3-487
IRET/IRETD/IRETQ—Interrupt Return 3-490
Jcc—Jump if Condition Is Met 3-499
JMP—Jump 3-504
KADDW/KADDB/KADDQ/KADDD—ADD Two Masks 3-513
KANDNW/KANDNB/KANDNQ/KANDND—Bitwise Logical AND NOT Masks 3-515
KANDW/KANDB/KANDQ/KANDD—Bitwise Logical AND Masks 3-516
KMOVW/KMOVB/KMOVQ/KMOVD—Move From and to Mask Registers 3-517
KNOTW/KNOTB/KNOTQ/KNOTD—NOT Mask Register 3-519
KORTESTW/KORTESTB/KORTESTQ/KORTESTD—OR Masks and Set Flags 3-520
KORW/KORB/KORQ/KORD—Bitwise Logical OR Masks 3-522
KSHIFTLW/KSHIFTLB/KSHIFTLQ/KSHIFTLD—Shift Left Mask Registers 3-523
KSHIFTRW/KSHIFTRB/KSHIFTRQ/KSHIFTRD—Shift Right Mask Registers 3-525
KTESTW/KTESTB/KTESTQ/KTESTD—Packed Bit Test Masks and Set Flags 3-527
KUNPCKBW/KUNPCKWD/KUNPCKDQ—Unpack for Mask Registers 3-529
KXNORW/KXNORB/KXNORQ/KXNORD—Bitwise Logical XNOR Masks 3-530
KXORW/KXORB/KXORQ/KXORD—Bitwise Logical XOR Masks 3-531
LAHF—Load Status Flags Into AH Register 3-532
LAR—Load Access Rights 3-533
LDDQU—Load Unaligned Integer 128 Bits 3-537
LDMXCSR—Load MXCSR Register 3-539
LDS/LES/LFS/LGS/LSS—Load Far Pointer 3-540
LDTILECFG—Load Tile Configuration 3-544
LEA—Load Effective Address 3-547
LEAVE—High Level Procedure Exit 3-550
LFENCE—Load Fence 3-552
LGDT/LIDT—Load Global/Interrupt Descriptor Table Register 3-553
LKGS—Load Kernel GS Base 3-556
LLDT—Load Local Descriptor Table Register 3-558
LMSW—Load Machine Status Word 3-560
LOADIWKEY—Load Internal Wrapping Key With Key Locker 3-562
LOCK—Assert LOCK# Signal Prefix 3-565

PAGE

LODS/LODSB/LODSW/LODSD/LODSQ—Load String 3-567
LOOP/LOOPcc—Loop According to ECX Counter 3-570
LSL—Load Segment Limit 3-573
LTR—Load Task Register 3-576
LZCNT—Count the Number of Leading Zero Bits 3-578

CHAPTER 4
INSTRUCTION SET REFERENCE, M-U
4.1 IMM8 CONTROL BYTE OPERATION FOR PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM 4-1
4.1.1 General Description. 4-1
4.1.2 Source Data Format 4-1
4.1.3 Aggregation Operation 4-2
4.1.4 Polarity 4-2
4.1.5 Output Selection 4-3
4.1.6 Valid/Invalid Override of Comparisons 4-3
4.1.7 Summary of Imm8 Control byte 4-4
4.1.8 Diagram Comparison and Aggregation Process 4-5
4.2 COMMON TRANSFORMATION AND PRIMITIVE FUNCTIONS FOR SHA1XXX AND SHA256XXX 4-5
4.3 INSTRUCTIONS (M-U) 4-6
MASKMOVDQU—Store Selected Bytes of Double Quadword 4-1
MASKMOVQ—Store Selected Bytes of Quadword 4-3
MAXPD—Maximum of Packed Double Precision Floating-Point Values 4-5
MAXPS—Maximum of Packed Single Precision Floating-Point Values 4-8
MAXSD—Return Maximum Scalar Double Precision Floating-Point Value 4-11
MAXSS—Return Maximum Scalar Single Precision Floating-Point Value 4-13
MFENCE—Memory Fence. 4-15
MINPD—Minimum of Packed Double Precision Floating-Point Values 4-16
MINPS—Minimum of Packed Single Precision Floating-Point Values 4-19
MINSD—Return Minimum Scalar Double Precision Floating-Point Value 4-22
MINSS—Return Minimum Scalar Single Precision Floating-Point Value 4-24
MONITOR—Set Up Monitor Address 4-26
MOV—Move 4-28
MOV—Move to/from Control Registers. 4-32
MOV—Move to/from Debug Registers 4-35
MOVAPD—Move Aligned Packed Double Precision Floating-Point Values 4-37
MOVAPS—Move Aligned Packed Single Precision Floating-Point Values 4-41
MOVBE—Move Data After Swapping Bytes 4-45
MOVDDUP—Replicate Double Precision Floating-Point Values 4-47
MOVDIR64B—Move 64 Bytes as Direct Store 4-50
MOVDIRI—Move Doubleword as Direct Store 4-52
MOVD/MOVQ—Move Doubleword/Move Quadword 4-54
MOVDQ2Q—Move Quadword from XMM to MMX Technology Register. 4-58
MOVDQA,VMOVDQA32/64—Move Aligned Packed Integer Values 4-59
MOVDQU,VMOVDQU8/16/32/64—Move Unaligned Packed Integer Values 4-64
MOVHLPS—Move Packed Single Precision Floating-Point Values High to Low 4-72
MOVHPD—Move High Packed Double Precision Floating-Point Value. 4-74
MOVHPS—Move High Packed Single Precision Floating-Point Values. 4-76
MOVLHPS—Move Packed Single Precision Floating-Point Values Low to High 4-78
MOVLPD—Move Low Packed Double Precision Floating-Point Value 4-80
MOVLPS—Move Low Packed Single Precision Floating-Point Values 4-82
MOVMSKPD—Extract Packed Double Precision Floating-Point Sign Mask 4-84
MOVMSKPS—Extract Packed Single Precision Floating-Point Sign Mask 4-86
MOVNTDQ—Store Packed Integers Using Non-Temporal Hint 4-88
MOVNTDQA—Load Double Quadword Non-Temporal Aligned Hint 4-90
MOVNTI—Store Doubleword Using Non-Temporal Hint 4-92
MOVNTPD—Store Packed Double Precision Floating-Point Values Using Non-Temporal Hint 4-94
MOVNTPS—Store Packed Single Precision Floating-Point Values Using Non-Temporal Hint 4-96
MOVNTQ—Store of Quadword Using Non-Temporal Hint. 4-98
MOVQ—Move Quadword 4-99
MOVQ2DQ—Move Quadword from MMX Technology to XMM Register. 4-102

PAGE

MOVS/MOVSB/MOVSW/MOVSD/MOVSQ—Move Data From String to String 4-104
MOVSD—Move or Merge Scalar Double Precision Floating-Point Value 4-108
MOVSHDUP—Replicate Single Precision Floating-Point Values....4-111
MOVSLDUP—Replicate Single Precision Floating-Point Values 4-114
MOVSS—Move or Merge Scalar Single Precision Floating-Point Value 4-117
MOVSX/MOVSXD—Move With Sign-Extension 4-120
MOVUPD—Move Unaligned Packed Double Precision Floating-Point Values....4-122
MOVUPS—Move Unaligned Packed Single Precision Floating-Point Values....4-126
MOVZX—Move With Zero-Extend....4-130
MPSADBW—Compute Multiple Packed Sums of Absolute Difference 4-132
MUL—Unsigned Multiply....4-140
MULPD—Multiply Packed Double Precision Floating-Point Values 4-142
MULPS—Multiply Packed Single Precision Floating-Point Values 4-145
MULSD—Multiply Scalar Double Precision Floating-Point Value 4-148
MULSS—Multiply Scalar Single Precision Floating-Point Values 4-150
MULX—Unsigned Multiply Without Affecting Flags....4-152
MWAIT—Monitor Wait....4-154
NEG—Two's Complement Negation 4-158
NOP—No Operation....4-160
NOT—One's Complement Negation 4-161
OR—Logical Inclusive OR 4-163
ORPD—Bitwise Logical OR of Packed Double Precision Floating-Point Values....4-165
ORPS—Bitwise Logical OR of Packed Single Precision Floating-Point Values....4-168
OUT—Output to Port....4-171
OUTS/OUTSB/OUTSW/OUTSD—Output String to Port....4-173
PABSB/PABSW/PABSD/PABSQ—Packed Absolute Value 4-177
PACKSSWB/PACKSSDW—Pack With Signed Saturation 4-183
PACKUSDW—Pack With Unsigned Saturation 4-191
PACKUSWB—Pack With Unsigned Saturation 4-196
PADDB/PADDW/PADDD/PADDQ—Add Packed Integers 4-201
PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation....4-208
PADDUSB/PADDUSW—Add Packed Unsigned Integers With Unsigned Saturation....4-212
PALIGNR—Packed Align Right 4-216
PAND—Logical AND....4-220
PANDN—Logical AND NOT....4-223
PAUSE—Spin Loop Hint 4-226
PAVGB/PAVGW—Average Packed Integers....4-227
PBLENDVB—Variable Blend Packed Bytes....4-231
PBLENDW—Blend Packed Words....4-235
PBNDKB—Platform Bind Key to Binary Large Object....4-238
PCLMULQDQ—Carry-Less Multiplication Quadword 4-242
PCMPEQB/PCMPEQW/PCMPEQD— Compare Packed Data for Equal 4-245
PCMPEQQ—Compare Packed Qword Data for Equal....4-251
PCMPESTRI—Packed Compare Explicit Length Strings, Return Index....4-254
PCMPESTRM—Packed Compare Explicit Length Strings, Return Mask 4-256
PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Integers for Greater Than....4-258
PCMPGTQ—Compare Packed Data for Greater Than 4-264
PCMPISTRI—Packed Compare Implicit Length Strings, Return Index 4-267
PCMPISTRM—Packed Compare Implicit Length Strings, Return Mask 4-269
PCONFIG—Platform Configuration 4-271
PDEP—Parallel Bits Deposit 4-282
PEXT—Parallel Bits Extract....4-284
PEXTRB/PEXTRD/PEXTRQ—Extract Byte/Dword/Qword....4-286
PEXTRW—Extract Word....4-289
PHADDSW—Packed Horizontal Add and Saturate....4-292
PHADDW/PHADDD—Packed Horizontal Add 4-294
PHMINPOSUW—Packed Horizontal Word Minimum....4-298
PHSUBSW—Packed Horizontal Subtract and Saturate....4-300
PHSUBW/PHSUBD—Packed Horizontal Subtract 4-302

PAGE

PINSRB/PINSRD/PINSRQ—Insert Byte/Dword/Qword 4-305
PINSRW—Insert Word 4-308
PMADDUBSW—Multiply and Add Packed Signed and Unsigned Bytes 4-310
PMADDWD—Multiply and Add Packed Integers 4-313
PMAXSB/PMAXSW/PMAXSD/PMAXSQ—Maximum of Packed Signed Integers 4-316
PMAXUB/PMAXUW—Maximum of Packed Unsigned Integers 4-323
PMAXUD/PMAXUQ—Maximum of Packed Unsigned Integers 4-328
PMINSB/PMINSW—Minimum of Packed Signed Integers 4-332
PMINSD/PMINSQ—Minimum of Packed Signed Integers 4-337
PMINUB/PMINUW—Minimum of Packed Unsigned Integers 4-341
PMINUD/PMINUQ—Minimum of Packed Unsigned Integers 4-346
PMOVMSKB—Move Byte Mask 4-350
PMOVSX—Packed Move With Sign Extend 4-352
PMOVZX—Packed Move With Zero Extend 4-362
PMULDQ—Multiply Packed Doubleword Integers 4-372
PMULHRSW—Packed Multiply High With Round and Scale 4-375
PMULHUW—Multiply Packed Unsigned Integers and Store High Result 4-379
PMULHW—Multiply Packed Signed Integers and Store High Result 4-383
PMULLD/PMULLQ—Multiply Packed Integers and Store Low Result 4-387
PMULLW—Multiply Packed Signed Integers and Store Low Result 4-391
PMULUDQ—Multiply Packed Unsigned Doubleword Integers 4-395
POP—Pop a Value From the Stack 4-398
POPA/POPAD—Pop All General-Purpose Registers 4-403
POPCNT—Return the Count of Number of Bits Set to 1 4-405
POPF/POPFD/POPFQ—Pop Stack Into EFLAGS Register 4-407
POR—Bitwise Logical OR 4-411
PREFETCHh—Prefetch Data Into Caches 4-414
PREFETCHW—Prefetch Data Into Caches in Anticipation of a Write 4-416
PSADBW—Compute Sum of Absolute Differences 4-418
PSHUFB—Packed Shuffle Bytes 4-422
PSHUFD—Shuffle Packed Doublewords 4-426
PSHUFHW—Shuffle Packed High Words 4-430
PSHUFLW—Shuffle Packed Low Words 4-433
PSHUFW—Shuffle Packed Words 4-436
PSIGNB/PSIGNW/PSIGND—Packed SIGN 4-437
PSLLDQ—Shift Double Quadword Left Logical 4-441
PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical 4-443
PSRAW/PSRAD/PSRAQ—Shift Packed Data Right Arithmetic 4-455
PSRLDQ—Shift Double Quadword Right Logical 4-465
PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical 4-467
PSUBB/PSUBW/PSUBD—Subtract Packed Integers 4-479
PSUBQ—Subtract Packed Quadword Integers 4-487
PSUBSB/PSUBSW—Subtract Packed Signed Integers With Signed Saturation 4-490
PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers With Unsigned Saturation 4-494
PTEST—Logical Compare 4-498
PTWRITE—Write Data to a Processor Trace Packet 4-500
PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ— Unpack High Data 4-502
PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ—Unpack Low Data 4-512
PUSH—Push Word, Doubleword, or Quadword Onto the Stack 4-522
PUSHA/PUSHAD—Push All General-Purpose Registers 4-526
PUSHF/PUSHFD/PUSHFQ—Push EFLAGS Register Onto the Stack 4-528
PXOR—Logical Exclusive OR 4-530
RCL/RCR/ROL/ROR—Rotate 4-533
RCPPS—Compute Reciprocals of Packed Single Precision Floating-Point Values 4-538
RCPSS—Compute Reciprocal of Scalar Single Precision Floating-Point Values 4-540
RDFSBASE/RDGSBASE—Read FS/GS Segment Base 4-542
RDMSR—Read From Model Specific Register 4-544
RDMSRLIST—Read List of Model Specific Registers 4-546
RDPID—Read Processor ID 4-548

PAGE

RDPKRU—Read Protection Key Rights for User Pages 4-549
RDPMC—Read Performance-Monitoring Counters 4-551
RDRAND—Read Random Number 4-554
RDSEED—Read Random SEED 4-556
RDSSPD/RDSSPQ—Read Shadow Stack Pointer 4-558
RDTSC—Read Time-Stamp Counter 4-559
RDTSCP—Read Time-Stamp Counter and Processor ID 4-561
REP—Repeat String Operation (Prefix) 4-563
REPE/REPZ—Repeat String Operation While Zero (Prefix) 4-565
REPNE/REPNZ—Repeat String Operation While Not Zero (Prefix) 4-567
RET—Return From Procedure 4-569
RORX — Rotate Right Logical Without Affecting Flags 4-582
ROUNDPD—Round Packed Double Precision Floating-Point Values 4-583
ROUNDPS—Round Packed Single Precision Floating-Point Values 4-586
ROUNDSD—Round Scalar Double Precision Floating-Point Values 4-588
ROUNDSS—Round Scalar Single Precision Floating-Point Values 4-590
RSM—Resume From System Management Mode 4-592
RSQRTPS—Compute Reciprocals of Square Roots of Packed Single Precision Floating-Point Values 4-594
RSQRTSS—Compute Reciprocal of Square Root of Scalar Single Precision Floating-Point Value 4-596
RSTORSSP—Restore Saved Shadow Stack Pointer 4-598
SAHF—Store AH Into Flags 4-601
SAL/SAR/SHL/SHR—Shift 4-603
SALC—Set AL Based on Carry Flag 4-608
SARX/SHLX/SHRX—Shift Without Affecting Flags 4-609
SAVEPREVSSP—Save Previous Shadow Stack Pointer 4-611
SBB—Integer Subtraction With Borrow 4-613
SCAS/SCASB/SCASW/SCASD—Scan String 4-616
SENDUIPI—Send User Interprocessor Interrupt 4-620
SERIALIZE—Serialize Instruction Execution 4-622
SETcc—Set Byte on Condition 4-623
SETSSBSY—Mark Shadow Stack Busy 4-626
SFENCE—Store Fence 4-628
SGDT—Store Global Descriptor Table Register 4-629
SHA1MSG1—Perform an Intermediate Calculation for the Next Four SHA1 Message Dwords 4-631
SHA1MSG2—Perform a Final Calculation for the Next Four SHA1 Message Dwords 4-632
SHA1NEXTE—Calculate SHA1 State Variable E After Four Rounds 4-633
SHA1RNDS4—Perform Four Rounds of SHA1 Operation 4-634
SHA256MSG1—Perform an Intermediate Calculation for the Next Four SHA256 Message Dwords 4-636
SHA256MSG2—Perform a Final Calculation for the Next Four SHA256 Message Dwords 4-637
SHA256RNDS2—Perform Two Rounds of SHA256 Operation 4-638
SHLD—Double Precision Shift Left 4-640
SHRD—Double Precision Shift Right 4-643
SHUFPD—Packed Interleave Shuffle of Pairs of Double Precision Floating-Point Values 4-646
SHUFPS—Packed Interleave Shuffle of Quadruplets of Single Precision Floating-Point Values 4-651
SIDT—Store Interrupt Descriptor Table Register 4-655
SLDT—Store Local Descriptor Table Register 4-657
SMSW—Store Machine Status Word 4-659
SQRTPD—Square Root of Double Precision Floating-Point Values 4-661
SQRTPS—Square Root of Single Precision Floating-Point Values 4-664
SQRTSD—Compute Square Root of Scalar Double Precision Floating-Point Value 4-667
SQRTSS—Compute Square Root of Scalar Single Precision Value 4-669
STAC—Set AC Flag in EFLAGS Register 4-671
STC—Set Carry Flag 4-672
STD—Set Direction Flag 4-673
STI—Set Interrupt Flag 4-674
STMXCSR—Store MXCSR Register State 4-676
STOS/STOSB/STOSW/STOSD/STOSQ—Store String 4-677
STR—Store Task Register 4-680
STTILECFG—Store Tile Configuration 4-682

STUI—Set User Interrupt Flag 4-684
SUB—Subtract 4-685
SUBPD—Subtract Packed Double Precision Floating-Point Values 4-687
SUBPS—Subtract Packed Single Precision Floating-Point Values. 4-690
SUBSD—Subtract Scalar Double Precision Floating-Point Value. 4-693
SUBSS—Subtract Scalar Single Precision Floating-Point Value. 4-695
SWAPGS—Swap GS Base Register 4-697
SYSCALL—Fast System Call 4-699
SYSENTER—Fast System Call. 4-702
SYSEXIT—Fast Return from Fast System Call 4-706
SYSRET—Return From Fast System Call 4-709
TCMMIMFP16PS/TCMMRLFP16PS—Matrix Multiplication of Complex Tiles Accumulated into Packed Single Precision Tile. 4-712
TDPBF16PS—Dot Product of BF16 Tiles Accumulated into Packed Single Precision Tile 4-715
TDPBSSD/TDPBSUD/TDPBUSD/TDPBUUD—Dot Product of Signed/Unsigned Bytes with Dword Accumulation 4-717
TDPFP16PS—Dot Product of FP16 Tiles Accumulated into Packed Single Precision Tile 4-719
TEST—Logical Compare 4-721
TESTUI—Determine User Interrupt Flag. 4-723
TILELOADD/TILELOADDT1—Load Tile. 4-724
TILERELEASE—Release Tile 4-726
TILESTORED—Store Tile 4-727
TILEZERO—Zero Tile. 4-728
TPAUSE—Timed PAUSE. 4-729
TZCNT—Count the Number of Trailing Zero Bits. 4-731
UCOMISD—Unordered Compare Scalar Double Precision Floating-Point Values and Set EFLAGS 4-733
UCOMISS—Unordered Compare Scalar Single Precision Floating-Point Values and Set EFLAGS 4-735
UD—Undefined Instruction 4-737
UIRET—User-Interrupt Return. 4-738
UMONITOR—User Level Set Up Monitor Address 4-740
UMWAIT—User Level Monitor Wait 4-742
UNPCKHPD—Unpack and Interleave High Packed Double Precision Floating-Point Values 4-744
UNPCKHPS—Unpack and Interleave High Packed Single Precision Floating-Point Values 4-748
UNPCKLPD—Unpack and Interleave Low Packed Double Precision Floating-Point Values. 4-752
UNPCKLPS—Unpack and Interleave Low Packed Single Precision Floating-Point Values. 4-756
URDMSR—User Read from Model-Specific Register 4-760
UWRMSR—User Write to Model-Specific Register. 4-762

CHAPTER 5
INSTRUCTION SET REFERENCE, V
5.1 TERNARY BIT VECTOR LOGIC TABLE. 5-1
5.2 INSTRUCTIONS (V). 5-3
VADDPH—Add Packed FP16 Values 5-1
VADDSH—Add Scalar FP16 Values 5-3
VALIGND/VALIGNQ—Align Doubleword/Quadword Vectors 5-4
VBCSTNEBF162PS—Load BF16 Element and Convert to FP32 Element With Broadcast 5-7
VBCSTNESH2PS—Load FP16 Element and Convert to FP32 Element with Broadcast. 5-8
VBLENDMPD/VBLENDMPS—Blend Float64/Float32 Vectors Using an OpMask Control. 5-9
VBROADCAST—Load with Broadcast Floating-Point Data 5-12
VCMPPH—Compare Packed FP16 Values. 5-20
VCMPSH—Compare Scalar FP16 Values. 5-22
VCOMISH—Compare Scalar Ordered FP16 Values and Set EFLAGS. 5-24
VCOMPRESSPD—Store Sparse Packed Double Precision Floating-Point Values Into Dense Memory. 5-26
VCOMPRESSPS—Store Sparse Packed Single Precision Floating-Point Values Into Dense Memory. 5-28
VCVTDQ2PH—Convert Packed Signed Doubleword Integers to Packed FP16 Values 5-30
VCVTNE2PS2BF16—Convert Two Packed Single Data to One Packed BF16 Data 5-32
VCVTNEEBF162PS—Convert Even Elements of Packed BF16 Values to FP32 Values 5-34
VCVTNEEPH2PS—Convert Even Elements of Packed FP16 Values to FP32 Values. 5-35
VCVTNEOBF162PS—Convert Odd Elements of Packed BF16 Values to FP32 Values 5-36
VCVTNEOPH2PS—Convert Odd Elements of Packed FP16 Values to FP32 Values. 5-37

PAGE

VCVTNEPS2BF16—Convert Packed Single Data to Packed BF16 Data 5-38
VCVTPD2PH—Convert Packed Double Precision FP Values to Packed FP16 Values 5-40
VCVTPD2QQ—Convert Packed Double Precision Floating-Point Values to Packed Quadword Integers 5-42
VCVTPD2UDQ—Convert Packed Double Precision Floating-Point Values to Packed Unsigned Doubleword Integers .. 5-44
VCVTPD2UQQ—Convert Packed Double Precision Floating-Point Values to Packed Unsigned Quadword Integers ... 5-47
VCVTPH2DQ—Convert Packed FP16 Values to Signed Doubleword Integers 5-50
VCVTPH2PD—Convert Packed FP16 Values to FP64 Values 5-52
VCVTPH2PS/VCVTPH2PSX—Convert Packed FP16 Values to Single Precision Floating-Point Values 5-54
VCVTPH2QQ—Convert Packed FP16 Values to Signed Quadword Integer Values. 5-58
VCVTPH2UDQ—Convert Packed FP16 Values to Unsigned Doubleword Integers 5-60
VCVTPH2UQQ—Convert Packed FP16 Values to Unsigned Quadword Integers. 5-62
VCVTPH2UW—Convert Packed FP16 Values to Unsigned Word Integers 5-64
VCVTPH2W—Convert Packed FP16 Values to Signed Word Integers. 5-66
VCVTPS2PH—Convert Single Precision FP Value to 16-bit FP Value 5-68
VCVTPS2PHX—Convert Packed Single Precision Floating-Point Values to Packed FP16 Values. 5-72
VCVTPS2QQ—Convert Packed Single Precision Floating-Point Values to Packed Signed Quadword Integer Values .. 5-74
VCVTPS2UDQ—Convert Packed Single Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values 5-76
VCVTPS2UQQ—Convert Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values 5-79
VCVTQQ2PD—Convert Packed Quadword Integers to Packed Double Precision Floating-Point Values 5-81
VCVTQQ2PH—Convert Packed Signed Quadword Integers to Packed FP16 Values 5-83
VCVTQQ2PS—Convert Packed Quadword Integers to Packed Single Precision Floating-Point Values 5-85
VCVTSD2SH—Convert Low FP64 Value to an FP16 Value 5-87
VCVTSD2USI—Convert Scalar Double Precision Floating-Point Value to Unsigned Integer 5-88
VCVTSH2SD—Convert Low FP16 Value to an FP64 Value 5-90
VCVTSH2SI—Convert Low FP16 Value to Signed Integer 5-91
VCVTSH2SS—Convert Low FP16 Value to FP32 Value. 5-93
VCVTSH2USI—Convert Low FP16 Value to Unsigned Integer 5-94
VCVTSI2SH—Convert a Signed Doubleword/Quadword Integer to an FP16 Value 5-96
VCVTSS2SH—Convert Low FP32 Value to an FP16 Value. 5-98
VCVTSS2USI—Convert Scalar Single Precision Floating-Point Value to Unsigned Doubleword Integer. 5-99
VCVTTPD2QQ—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Quadword Integers. 5-101
VCVTTPD2UDQ—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Unsigned Doubleword Integers. 5-103
VCVTTPD2UQQ—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Unsigned Quadword Integers 5-105
VCVTTPH2DQ—Convert with Truncation Packed FP16 Values to Signed Doubleword Integers. 5-107
VCVTTPH2QQ—Convert with Truncation Packed FP16 Values to Signed Quadword Integers 5-109
VCVTTPH2UDQ—Convert with Truncation Packed FP16 Values to Unsigned Doubleword Integers. 5-111
VCVTTPH2UQQ—Convert with Truncation Packed FP16 Values to Unsigned Quadword Integers 5-113
VCVTTPH2UW—Convert Packed FP16 Values to Unsigned Word Integers 5-115
VCVTTPH2W—Convert Packed FP16 Values to Signed Word Integers 5-117
VCVTTPS2QQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Signed Quadword Integer Values 5-119
VCVTTPS2UDQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values 5-121
VCVTTPS2UQQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values 5-123
VCVTTSD2USI—Convert With Truncation Scalar Double Precision Floating-Point Value to Unsigned Integer 5-125
VCVTTSH2SI—Convert with Truncation Low FP16 Value to a Signed Integer 5-127
VCVTTSH2USI—Convert with Truncation Low FP16 Value to an Unsigned Integer 5-128
VCVTTSS2USI—Convert With Truncation Scalar Single Precision Floating-Point Value to Unsigned Integer. 5-129
VCVTUDQ2PD—Convert Packed Unsigned Doubleword Integers to Packed Double Precision Floating-Point Values. 5-131
VCVTUDQ2PH—Convert Packed Unsigned Doubleword Integers to Packed FP16 Values. 5-133
VCVTUDQ2PS—Convert Packed Unsigned Doubleword Integers to Packed Single Precision Floating-Point Values .. 5-135
VCVTUQQ2PD—Convert Packed Unsigned Quadword Integers to Packed Double Precision Floating-Point Values .. 5-138
VCVTUQQ2PH—Convert Packed Unsigned Quadword Integers to Packed FP16 Values 5-140

PAGE

VCVTUQQ2PS—Convert Packed Unsigned Quadword Integers to Packed Single Precision Floating-Point Values ... 5-142
VCVTUSI2SD—Convert Unsigned Integer to Scalar Double Precision Floating-Point Value ... 5-144
VCVTUSI2SH—Convert Unsigned Doubleword Integer to an FP16 Value ... 5-146
VCVTUSI2SS—Convert Unsigned Integer to Scalar Single Precision Floating-Point Value ... 5-148
VCVTUW2PH—Convert Packed Unsigned Word Integers to FP16 Values ... 5-150
VCVTW2PH—Convert Packed Signed Word Integers to FP16 Values ... 5-152
VDBPSADBW—Double Block Packed Sum-Absolute-Differences (SAD) on Unsigned Bytes ... 5-154
VDIVPH—Divide Packed FP16 Values ... 5-157
VDIVSH—Divide Scalar FP16 Values ... 5-159
VDPBF16PS—Dot Product of BF16 Pairs Accumulated Into Packed Single Precision ... 5-160
VERR/VERW—Verify a Segment for Reading or Writing ... 5-162
VEXPANDPD—Load Sparse Packed Double Precision Floating-Point Values From Dense Memory ... 5-164
VEXPANDPS—Load Sparse Packed Single Precision Floating-Point Values From Dense Memory ... 5-166
VEXTRACTF128/VEXTRACTF32x4/VEXTRACTF64x2/VEXTRACTF32x8/VEXTRACTF64x4— Extract Packed Floating-Point Values ... 5-168
VEXTRACTI128/VEXTRACTI32x4/VEXTRACTI64x2/VEXTRACTI32x8/VEXTRACTI64x4—Extract Packed Integer Values ... 5-174
VFCMADDCPH/VFMADDCPH—Complex Multiply and Accumulate FP16 Values ... 5-180
VFCMADDCSH/VFMADDCSH—Complex Multiply and Accumulate Scalar FP16 Values ... 5-183
VFCMULCPH/VFMULCPH—Complex Multiply FP16 Values ... 5-185
VFCMULCSH/VFMULCSH—Complex Multiply Scalar FP16 Values ... 5-189
VFIXUPIMMPD—Fix Up Special Packed Float64 Values ... 5-191
VFIXUPIMMPS—Fix Up Special Packed Float32 Values ... 5-195
VFIXUPIMMSD—Fix Up Special Scalar Float64 Value ... 5-199
VFIXUPIMMSS—Fix Up Special Scalar Float32 Value ... 5-203
VFMADD132PD/VFMADD213PD/VFMADD231PD—Fused Multiply-Add of Packed Double Precision Floating-Point Values ... 5-207
VF[,N]MADD[132,213,231]PH—Fused Multiply-Add of Packed FP16 Values ... 5-214
VFMADD132PS/VFMADD213PS/VFMADD231PS—Fused Multiply-Add of Packed Single Precision Floating-Point Values ... 5-220
VFMADD132SD/VFMADD213SD/VFMADD231SD—Fused Multiply-Add of Scalar Double Precision Floating-Point Values ... 5-226
VF[,N]MADD[132,213,231]SH—Fused Multiply-Add of Scalar FP16 Values ... 5-229
VFMADD132SS/VFMADD213SS/VFMADD231SS—Fused Multiply-Add of Scalar Single Precision Floating-Point Values ... 5-232
VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD—Fused Multiply-Alternating Add/Subtract of Packed Double Precision Floating-Point Values ... 5-235
VFMADDSUB132PH/VFMADDSUB213PH/VFMADDSUB231PH—Fused Multiply-Alternating Add/Subtract of Packed FP16 Values ... 5-243
VFMADDSUB132PS/VFMADDSUB213PS/VFMADDSUB231PS—Fused Multiply-Alternating Add/Subtract of Packed Single Precision Floating-Point Values ... 5-248
VFMSUB132PD/VFMSUB213PD/VFMSUB231PD—Fused Multiply-Subtract of Packed Double Precision Floating-Point Values ... 5-256
VF[,N]MSUB[132,213,231]PH—Fused Multiply-Subtract of Packed FP16 Values ... 5-263
VFMSUB132PS/VFMSUB213PS/VFMSUB231PS—Fused Multiply-Subtract of Packed Single Precision Floating-Point Values ... 5-269
VFMSUB132SD/VFMSUB213SD/VFMSUB231SD—Fused Multiply-Subtract of Scalar Double Precision Floating-Point Values ... 5-276
VF[,N]MSUB[132,213,231]SH—Fused Multiply-Subtract of Scalar FP16 Values ... 5-279
VFMSUB132SS/VFMSUB213SS/VFMSUB231SS—Fused Multiply-Subtract of Scalar Single Precision Floating-Point Values ... 5-282
VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD—Fused Multiply-Alternating Subtract/Add of Packed Double Precision Floating-Point Values ... 5-285
VFMSUBADD132PH/VFMSUBADD213PH/VFMSUBADD231PH—Fused Multiply-Alternating Subtract/Add of Packed FP16 Values ... 5-292
VFMSUBADD132PS/VFMSUBADD213PS/VFMSUBADD231PS—Fused Multiply-Alternating Subtract/Add of Packed Single Precision Floating-Point Values ... 5-297
VFNMADD132PD/VFNMADD213PD/VFNMADD231PD—Fused Negative Multiply-Add of Packed Double Precision Floating-Point Values ... 5-305

PAGE

VFNMADD132PS/VFNMADD213PS/VFNMADD231PS—Fused Negative Multiply-Add of Packed Single Precision Floating-Point Values 5-312
VFNMADD132SD/VFNMADD213SD/VFNMADD231SD—Fused Negative Multiply-Add of Scalar Double Precision Floating-Point Values 5-319
VFNMADD132SS/VFNMADD213SS/VFNMADD231SS—Fused Negative Multiply-Add of Scalar Single Precision Floating-Point Values 5-322
VFNMSUB132PD/VFNMSUB213PD/VFNMSUB231PD—Fused Negative Multiply-Subtract of Packed Double Precision Floating-Point Values 5-325
VFNMSUB132PS/VFNMSUB213PS/VFNMSUB231PS—Fused Negative Multiply-Subtract of Packed Single Precision Floating-Point Values 5-332
VFNMSUB132SD/VFNMSUB213SD/VFNMSUB231SD—Fused Negative Multiply-Subtract of Scalar Double Precision Floating-Point Values 5-339
VFNMSUB132SS/VFNMSUB213SS/VFNMSUB231SS—Fused Negative Multiply-Subtract of Scalar Single Precision Floating-Point Values 5-342
VFPCLASSPD—Tests Types of Packed Float64 Values 5-345
VFPCLASSPH—Test Types of Packed FP16 Values 5-348
VFPCLASSPS—Tests Types of Packed Float32 Values 5-351
VFPCLASSSD—Tests Type of a Scalar Float64 Value 5-353
VFPCLASSSH—Test Types of Scalar FP16 Values 5-355
VFPCLASSSS—Tests Type of a Scalar Float32 Value 5-356
VGATHERDPD/VGATHERQPD—Gather Packed Double Precision Floating-Point Values Using Signed Dword/Qword Indices 5-358
VGATHERDPS/VGATHERDPD—Gather Packed Single, Packed Double with Signed Dword Indices 5-362
VGATHERDPS/VGATHERQPS—Gather Packed Single Precision Floating-Point Values Using Signed Dword/Qword Indices 5-365
VGATHERQPS/VGATHERQPD—Gather Packed Single, Packed Double with Signed Qword Indices 5-369
VGETEXPPD—Convert Exponents of Packed Double Precision Floating-Point Values to Double Precision Floating-Point Values 5-372
VGETEXPPH—Convert Exponents of Packed FP16 Values to FP16 Values 5-376
VGETEXPPS—Convert Exponents of Packed Single Precision Floating-Point Values to Single Precision Floating-Point Values 5-379
VGETEXPSD—Convert Exponents of Scalar Double Precision Floating-Point Value to Double Precision Floating-Point Value 5-383
VGETEXPSH—Convert Exponents of Scalar FP16 Values to FP16 Values 5-385
VGETEXPSS—Convert Exponents of Scalar Single Precision Floating-Point Value to Single Precision Floating-Point Value 5-387
VGETMANTPD—Extract Float64 Vector of Normalized Mantissas From Float64 Vector 5-389
VGETMANTPH—Extract FP16 Vector of Normalized Mantissas from FP16 Vector 5-393
VGETMANTPS—Extract Float32 Vector of Normalized Mantissas From Float32 Vector 5-397
VGETMANTSD—Extract Float64 of Normalized Mantissa From Float64 Scalar 5-400
VGETMANTSH—Extract FP16 of Normalized Mantissa from FP16 Scalar 5-402
VGETMANTSS—Extract Float32 Vector of Normalized Mantissa From Float32 Scalar 5-404
VINSERTF128/VINSERTF32x4/VINSERTF64x2/VINSERTF32x8/VINSERTF64x4—Insert Packed Floating-Point Values 5-406
VINSERTI128/VINSERTI32x4/VINSERTI64x2/VINSERTI32x8/VINSERTI64x4—Insert Packed Integer Values 5-411
VMASKMOV—Conditional SIMD Packed Loads and Stores 5-416
VMAXPH—Return Maximum of Packed FP16 Values 5-419
VMAXSH—Return Maximum of Scalar FP16 Values 5-421
VMINPH—Return Minimum of Packed FP16 Values 5-423
VMINSH—Return Minimum Scalar FP16 Value 5-425
VMOVSH—Move Scalar FP16 Value 5-427
VMOVW—Move Word 5-429
VMULPH—Multiply Packed FP16 Values 5-430
VMULSH—Multiply Scalar FP16 Values 5-432
VP2INTERSECTD/VP2INTERSECTQ—Compute Intersection Between DWORDS/QUADWORDS to a Pair of Mask Registers 5-433
VPBLENDD—Blend Packed Dwords 5-435
VPBLENDMB/VPBLENDMW—Blend Byte/Word Vectors Using an Opmask Control 5-437
VPBLENDMD/VPBLENDMQ—Blend Int32/Int64 Vectors Using an OpMask Control 5-439

PAGE

VPBROADCAST—Load Integer and Broadcast 5-442
VPBROADCASTB/W/D/Q—Load With Broadcast Integer Data From General Purpose Register 5-451
VPBROADCASTM—Broadcast Mask to Vector Register 5-454
VPCMPB/VPCMPUB—Compare Packed Byte Values Into Mask 5-456
VPCMPD/VPCMPUD—Compare Packed Integer Values Into Mask 5-459
VPCMPQ/VPCMPUQ—Compare Packed Integer Values Into Mask 5-462
VPCMPW/VPCMPUW—Compare Packed Word Values Into Mask 5-465
VPCOMPRESSB/VCOMPRESSW—Store Sparse Packed Byte/Word Integer Values Into Dense Memory/Register 5-468
VPCOMPRESSD—Store Sparse Packed Doubleword Integer Values Into Dense Memory/Register 5-471
VPCOMPRESSQ—Store Sparse Packed Quadword Integer Values Into Dense Memory/Register 5-473
VPCONFLICTD/Q—Detect Conflicts Within a Vector of Packed Dword/Qword Values Into Dense Memory/ Register. 5-475
VPDPB[SU,UU,SS]D[,S]—Multiply and Add Unsigned and Signed Bytes With and Without Saturation 5-478
VPDPBUSD—Multiply and Add Unsigned and Signed Bytes 5-481
VPDPBUSDS—Multiply and Add Unsigned and Signed Bytes With Saturation 5-484
VPDPWSSD—Multiply and Add Signed Word Integers 5-487
VPDPWSSDS—Multiply and Add Signed Word Integers With Saturation 5-489
VPDPW[SU,US,UU]D[,S]—Multiply and Add Unsigned and Signed Words With and Without Saturation 5-491
VPERM2F128—Permute Floating-Point Values. 5-494
VPERM2I128—Permute Integer Values 5-496
VPERMB—Permute Packed Bytes Elements 5-498
VPERMD/VPERMW—Permute Packed Doubleword/Word Elements 5-500
VPERMI2B—Full Permute of Bytes From Two Tables Overwriting the Index 5-503
VPERMI2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting the Index 5-505
VPERMILPD—Permute In-Lane of Pairs of Double Precision Floating-Point Values 5-511
VPERMILPS—Permute In-Lane of Quadruples of Single Precision Floating-Point Values 5-517
VPERMPD—Permute Double Precision Floating-Point Elements 5-522
VPERMPS—Permute Single Precision Floating-Point Elements. 5-526
VPERMQ—Qwords Element Permutation. 5-529
VPERMT2B—Full Permute of Bytes From Two Tables Overwriting a Table. 5-533
VPERMT2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting One Table 5-535
VPEXPANDB/VPEXPANDW—Expand Byte/Word Values 5-541
VPEXPANDD—Load Sparse Packed Doubleword Integer Values From Dense Memory/Register. 5-544
VPEXPANDQ—Load Sparse Packed Quadword Integer Values From Dense Memory/Register 5-546
VPGATHERDD/VPGATHERDQ—Gather Packed Dword, Packed Qword With Signed Dword Indices 5-548
VPGATHERDD/VPGATHERQD—Gather Packed Dword Values Using Signed Dword/Qword Indices 5-551
VPGATHERDQ/VPGATHERQQ—Gather Packed Qword Values Using Signed Dword/Qword Indices 5-555
VPGATHERQD/VPGATHERQQ—Gather Packed Dword, Packed Qword with Signed Qword Indices 5-559
VPLZCNTD/Q—Count the Number of Leading Zero Bits for Packed Dword, Packed Qword Values 5-562
VPMADD52HUQ—Packed Multiply of Unsigned 52-Bit Unsigned Integers and Add High 52-Bit Products to 64-Bit Accumulators 5-565
VPMADD52LUQ—Packed Multiply of Unsigned 52-Bit Integers and Add the Low 52-Bit Products to Qword Accumulators 5-568
VPMASKMOV—Conditional SIMD Integer Packed Loads and Stores 5-571
VPMOVB2M/VPMOVW2M/VPMOVD2M/VPMOVQ2M—Convert a Vector Register to a Mask 5-574
VPMOVDB/VPMOVSDB/VPMOVUSDB—Down Convert DWord to Byte 5-577
VPMOVDW/VPMOVSDW/VPMOVUSDW—Down Convert DWord to Word 5-581
VPMOVM2B/VPMOVM2W/VPMOVM2D/VPMOVM2Q—Convert a Mask Register to a Vector Register 5-585
VPMOVQB/VPMOVSQB/VPMOVUSQB—Down Convert QWord to Byte 5-588
VPMOVQD/VPMOVSQD/VPMOVUSQD—Down Convert QWord to DWord 5-592
VPMOVQW/VPMOVSQW/VPMOVUSQW—Down Convert QWord to Word 5-596
VPMOVWB/VPMOVSWB/VPMOVUSWB—Down Convert Word to Byte 5-600
VPMULTISHIFTQB—Select Packed Unaligned Bytes From Quadword Sources 5-604
VPOPCNT—Return the Count of Number of Bits Set to 1 in BYTE/WORD/DWORD/QWORD. 5-606
VPROLD/VPROLVD/VPROLQ/VPROLVQ—Bit Rotate Left 5-610
VPRORD/VPRORVD/VPRORQ/VPRORVQ—Bit Rotate Right 5-614
VPSCATTERDD/VPSCATTERDQ/VPSCATTERQD/VPSCATTERQQ—Scatter Packed Dword, Packed Qword with Signed Dword, Signed Qword Indices 5-618
VPSHLD—Concatenate and Shift Packed Data Left Logical 5-622
VPSHLDV—Concatenate and Variable Shift Packed Data Left Logical 5-625

PAGE

VPSHRD—Concatenate and Shift Packed Data Right Logical 5-628
VPSHRDV—Concatenate and Variable Shift Packed Data Right Logical 5-631
VPSHUFBITQMB—Shuffle Bits From Quadword Elements Using Byte Indexes Into Mask 5-634
VPSLLVW/VPSLLVD/VPSLLVQ—Variable Bit Shift Left Logical 5-636
VPSRAVW/VPSRAVD/VPSRAVQ—Variable Bit Shift Right Arithmetic 5-641
VPSRLVW/VPSRLVD/VPSRLVQ—Variable Bit Shift Right Logical 5-646
VPTERNLOGD/VPTERNLOGQ—Bitwise Ternary Logic 5-651
VPTESTMB/VPTESTMW/VPTESTMD/VPTESTMQ—Logical AND and Set Mask 5-654
VPTESTNMB/W/D/Q—Logical NAND and Set 5-657
VRANGEPD—Range Restriction Calculation for Packed Pairs of Float64 Values 5-661
VRANGEPS—Range Restriction Calculation for Packed Pairs of Float32 Values 5-665
VRANGESD—Range Restriction Calculation From a Pair of Scalar Float64 Values 5-668
VRANGESS—Range Restriction Calculation From a Pair of Scalar Float32 Values 5-671
VRCP14PD—Compute Approximate Reciprocals of Packed Float64 Values 5-674
VRCP14PS—Compute Approximate Reciprocals of Packed Float32 Values 5-676
VRCP14SD—Compute Approximate Reciprocal of Scalar Float64 Value 5-678
VRCP14SS—Compute Approximate Reciprocal of Scalar Float32 Value 5-680
VRCPPH—Compute Reciprocals of Packed FP16 Values 5-682
VRCPSH—Compute Reciprocal of Scalar FP16 Value 5-684
VREDUCEPD—Perform Reduction Transformation on Packed Float64 Values 5-685
VREDUCEPH—Perform Reduction Transformation on Packed FP16 Values 5-688
VREDUCEPS—Perform Reduction Transformation on Packed Float32 Values 5-691
VREDUCESD—Perform a Reduction Transformation on a Scalar Float64 Value 5-693
VREDUCESH—Perform Reduction Transformation on Scalar FP16 Value 5-695
VREDUCESS—Perform a Reduction Transformation on a Scalar Float32 Value 5-697
VRNDSCALEPD—Round Packed Float64 Values to Include a Given Number of Fraction Bits 5-699
VRNDSCALEPH—Round Packed FP16 Values to Include a Given Number of Fraction Bits 5-702
VRNDSCALEPS—Round Packed Float32 Values to Include a Given Number of Fraction Bits 5-705
VRNDSCALESD—Round Scalar Float64 Value to Include a Given Number of Fraction Bits 5-708
VRNDSCALESH—Round Scalar FP16 Value to Include a Given Number of Fraction Bits 5-710
VRNDSCALESS—Round Scalar Float32 Value to Include a Given Number of Fraction Bits 5-712
VRSQRT14PD—Compute Approximate Reciprocals of Square Roots of Packed Float64 Values 5-714
VRSQRT14PS—Compute Approximate Reciprocals of Square Roots of Packed Float32 Values 5-716
VRSQRT14SD—Compute Approximate Reciprocal of Square Root of Scalar Float64 Value 5-718
VRSQRT14SS—Compute Approximate Reciprocal of Square Root of Scalar Float32 Value 5-720
VRSQRTPH—Compute Reciprocals of Square Roots of Packed FP16 Values 5-722
VRSQRTSH—Compute Approximate Reciprocal of Square Root of Scalar FP16 Value 5-724
VSCALEFPD—Scale Packed Float64 Values With Float64 Values 5-725
VSCALEFPH—Scale Packed FP16 Values with FP16 Values 5-728
VSCALEFPS—Scale Packed Float32 Values With Float32 Values 5-731
VSCALEFSD—Scale Scalar Float64 Values With Float64 Values 5-734
VSCALEFSH—Scale Scalar FP16 Values with FP16 Values 5-736
VSCALEFSS—Scale Scalar Float32 Value With Float32 Value 5-738
VSCATTERDPS/VSCATTERDPD/VSCATTERQPS/VSCATTERQPD—Scatter Packed Single Precision, Packed Double Precision Floating-Point Values with Signed Dword and Qword Indices 5-740
VSHA512MSG1—Perform an Intermediate Calculation for the Next Four SHA512 Message Qwords 5-744
VSHA512MSG2—Perform a Final Calculation for the Next Four SHA512 Message Qwords 5-746
VSHA512RNDS2—Perform Two Rounds of SHA512 Operation 5-748
VSHUFF32x4/VSHUFF64x2/VSHUFI32x4/VSHUFI64x2—Shuffle Packed Values at 128-Bit Granularity 5-750
VSM3MSG1—Perform Initial Calculation for the Next Four SM3 Message Words 5-755
VSM3MSG2—Perform Final Calculation for the Next Four SM3 Message Words 5-757
VSM3RNDS2—Perform Two Rounds of SM3 Operation 5-759
VSM4KEY4—Perform Four Rounds of SM4 Key Expansion 5-761
VSM4RNDS4—Performs Four Rounds of SM4 Encryption 5-763
VSQRTPH—Compute Square Root of Packed FP16 Values 5-765
VSQRTSH—Compute Square Root of Scalar FP16 Value 5-767
VSUBPH—Subtract Packed FP16 Values 5-768
VSUBSH—Subtract Scalar FP16 Value 5-770
VTESTPD/VTESTPS—Packed Bit Test 5-771

VUCOMISH—Unordered Compare Scalar FP16 Values and Set EFLAGS 5-774
VZEROALL—Zero XMM, YMM, and ZMM Registers 5-775
VZEROUPPER—Zero Upper Bits of YMM and ZMM Registers 5-776

CHAPTER 6
INSTRUCTION SET REFERENCE, W-Z
6.1 INSTRUCTIONS (W-Z) 6-1
WAIT/FWAIT—Wait 6-1
WBINVD—Write Back and Invalidate Cache 6-2
WBNOINVD—Write Back and Do Not Invalidate Cache 6-4
WRFSBASE/WRGSBASE—Write FS/GS Segment Base 6-6
WRMSR—Write to Model Specific Register 6-8
WRMSRLIST—Write List of Model Specific Registers 6-10
WRMSRNS—Non-Serializing Write to Model Specific Register 6-12
WRPKRU—Write Data to User Page Key Register 6-15
WRSSD/WRSSQ—Write to Shadow Stack 6-17
WRUSSD/WRUSSQ—Write to User Shadow Stack 6-19
XABORT—Transactional Abort 6-21
XACQUIRE/XRELEASE—Hardware Lock Elision Prefix Hints 6-23
XADD—Exchange and Add 6-27
XBEGIN—Transactional Begin 6-29
XCHG—Exchange Register/Memory With Register 6-32
XEND—Transactional End 6-34
XGETBV—Get Value of Extended Control Register 6-36
XLAT/XLATB—Table Look-up Translation 6-38
XOR—Logical Exclusive OR 6-40
XORPD—Bitwise Logical XOR of Packed Double Precision Floating-Point Values 6-42
XORPS—Bitwise Logical XOR of Packed Single Precision Floating-Point Values 6-45
XRESLDTRK—Resume Tracking Load Addresses 6-48
XRSTOR—Restore Processor Extended States 6-49
XRSTORS—Restore Processor Extended States Supervisor 6-54
XSAVE—Save Processor Extended States 6-58
XSAVEC—Save Processor Extended States With Compaction 6-61
XSAVEOPT—Save Processor Extended States Optimized 6-64
XSAVES—Save Processor Extended States Supervisor 6-67
XSETBV—Set Extended Control Register 6-70
XSUSLDTRK—Suspend Tracking Load Addresses 6-72
XTEST—Test if in Transactional Execution 6-73

CHAPTER 7
SAFER MODE EXTENSIONS REFERENCE
7.1 OVERVIEW 7-1
7.2 SMX FUNCTIONALITY 7-1
7.2.1 Detecting and Enabling SMX 7-1
7.2.2 SMX Instruction Summary 7-2
7.2.2.1 GETSEC[CAPABILITIES] 7-3
7.2.2.2 GETSEC[ENTERACCS] 7-3
7.2.2.3 GETSEC[EXITAC] 7-3
7.2.2.4 GETSEC[SENTER] 7-4
7.2.2.5 GETSEC[SEXIT] 7-4
7.2.2.6 GETSEC[PARAMETERS] 7-4
7.2.2.7 GETSEC[SMCTRL] 7-4
7.2.2.8 GETSEC[WAKEUP] 7-4
7.2.3 Measured Environment and SMX 7-5
7.3 GETSEC LEAF FUNCTIONS 7-5
GETSEC[CAPABILITIES]—Report the SMX Capabilities 7-7
GETSEC[ENTERACCS]—Execute Authenticated Chipset Code 7-10
GETSEC[EXITAC]—Exit Authenticated Code Execution Mode 7-18
GETSEC[SENTER]—Enter a Measured Environment 7-22
GETSEC[SEXIT]—Exit Measured Environment 7-31

PAGE

GETSEC[PARAMETERS]—Report the SMX Parameters 7-34
GETSEC[SMCTRL]—SMX Mode Control 7-38
GETSEC[WAKEUP]—Wake Up Sleeping Processors in Measured Environment 7-41

CHAPTER 8
INSTRUCTION SET REFERENCE UNIQUE TO INTEL® XEON PHI™ PROCESSORS
PREFETCHWT1—Prefetch Vector Data Into Caches With Intent to Write and T1 Hint 8-2
V4FMADDPS/V4FNMADDPS—Packed Single Precision Floating-Point Fused Multiply-Add (4-Iterations) 8-4
V4FMADDSS/V4FNMADDSS—Scalar Single Precision Floating-Point Fused Multiply-Add (4-Iterations) 8-6
VEXP2PD—Approximation to the Exponential 2^x of Packed Double Precision Floating-Point Values With Less Than 2^-23 Relative Error 8-8
VEXP2PS—Approximation to the Exponential 2^x of Packed Single Precision Floating-Point Values With Less Than 2^-23 Relative Error 8-10
VGATHERPF0DPS/VGATHERPF0QPS/VGATHERPF0DPD/VGATHERPF0QPD—Sparse Prefetch Packed SP/DP Data Values With Signed Dword, Signed Qword Indices Using T0 Hint 8-12
VGATHERPF1DPS/VGATHERPF1QPS/VGATHERPF1DPD/VGATHERPF1QPD—Sparse Prefetch Packed SP/DP Data Values With Signed Dword, Signed Qword Indices Using T1 Hint 8-14
VP4DPWSSDS—Dot Product of Signed Words With Dword Accumulation and Saturation (4-Iterations) 8-16
VP4DPWSSD—Dot Product of Signed Words With Dword Accumulation (4-Iterations) 8-18
VRCP28PD—Approximation to the Reciprocal of Packed Double Precision Floating-Point Values With Less Than 2^-28 Relative Error 8-20
VRCP28SD—Approximation to the Reciprocal of Scalar Double Precision Floating-Point Value With Less Than 2^-28 Relative Error 8-22
VRCP28PS—Approximation to the Reciprocal of Packed Single Precision Floating-Point Values With Less Than 2^-28 Relative Error 8-24
VRCP28SS—Approximation to the Reciprocal of Scalar Single Precision Floating-Point Value With Less Than 2^-28 Relative Error 8-26
VRSQRT28PD—Approximation to the Reciprocal Square Root of Packed Double Precision Floating-Point Values With Less Than 2^-28 Relative Error 8-28
VRSQRT28SD—Approximation to the Reciprocal Square Root of Scalar Double Precision Floating-Point Value With Less Than 2^-28 Relative Error 8-30
VRSQRT28PS—Approximation to the Reciprocal Square Root of Packed Single Precision Floating-Point Values With Less Than 2^-28 Relative Error 8-32
VRSQRT28SS—Approximation to the Reciprocal Square Root of Scalar Single Precision Floating-Point Value With Less Than 2^-28 Relative Error 8-34
VSCATTERPF0DPS/VSCATTERPF0QPS/VSCATTERPF0DPD/VSCATTERPF0QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T0 Hint With Intent to Write 8-36
VSCATTERPF1DPS/VSCATTERPF1QPS/VSCATTERPF1DPD/VSCATTERPF1QPD—Sparse Prefetch Packed SP/DP Data Values With Signed Dword, Signed Qword Indices Using T1 Hint With Intent to Write 8-38

APPENDIX A
OPCODE MAP
A.1 USING OPCODE TABLES A-1
A.2 KEY TO ABBREVIATIONS A-1
A.2.1 Codes for Addressing Method A-1
A.2.2 Codes for Operand Type A-2
A.2.3 Register Codes A-3
A.2.4 Opcode Look-up Examples for One, Two, and Three-Byte Opcodes A-3
A.2.4.1 One-Byte Opcode Instructions A-3
A.2.4.2 Two-Byte Opcode Instructions A-4
A.2.4.3 Three-Byte Opcode Instructions A-5
A.2.4.4 VEX Prefix Instructions A-5
A.2.5 Superscripts Utilized in Opcode Tables A-6
A.3 ONE, TWO, AND THREE-BYTE OPCODE MAPS A-6
A.4 OPCODE EXTENSIONS FOR ONE-BYTE AND TWO-BYTE OPCODES A-17
A.4.1 Opcode Look-up Examples Using Opcode Extensions A-17
A.4.2 Opcode Extension Tables A-17
A.5 ESCAPE OPCODE INSTRUCTIONS A-21
A.5.1 Opcode Look-up Examples for Escape Instruction Opcodes A-21
A.5.2 Escape Opcode Instruction Tables A-21
A.5.2.1 Escape Opcodes with D8 as First Byte A-21

A.5.2.2 Escape Opcodes with D9 as First Byte ... A-22
A.5.2.3 Escape Opcodes with DA as First Byte ... A-23
A.5.2.4 Escape Opcodes with DB as First Byte ... A-24
A.5.2.5 Escape Opcodes with DC as First Byte ... A-25
A.5.2.6 Escape Opcodes with DD as First Byte ... A-26
A.5.2.7 Escape Opcodes with DE as First Byte ... A-27
A.5.2.8 Escape Opcodes with DF As First Byte ... A-28

APPENDIX B
INSTRUCTION FORMATS AND ENCODINGS
B.1 MACHINE INSTRUCTION FORMAT ... B-1
B.1.1 Legacy Prefixes ... B-1
B.1.2 REX Prefixes ... B-2
B.1.3 Opcode Fields ... B-2
B.1.4 Special Fields ... B-2
B.1.4.1 Reg Field (reg) for Non-64-Bit Modes ... B-3
B.1.4.2 Reg Field (reg) for 64-Bit Mode ... B-4
B.1.4.3 Encoding of Operand Size (w) Bit ... B-4
B.1.4.4 Sign-Extend (s) Bit ... B-5
B.1.4.5 Segment Register (sreg) Field ... B-5
B.1.4.6 Special-Purpose Register (eee) Field ... B-5
B.1.4.7 Condition Test (tttn) Field ... B-6
B.1.4.8 Direction (d) Bit ... B-6
B.1.5 Other Notes ... B-6
B.2 GENERAL-PURPOSE INSTRUCTION FORMATS AND ENCODINGS FOR NON-64-BIT MODES ... B-7
B.2.1 General Purpose Instruction Formats and Encodings for 64-Bit Mode ... B-18
B.3 PENTIUM® PROCESSOR FAMILY INSTRUCTION FORMATS AND ENCODINGS ... B-37
B.4 64-BIT MODE INSTRUCTION ENCODINGS FOR SIMD INSTRUCTION EXTENSIONS ... B-37
B.5 MMX INSTRUCTION FORMATS AND ENCODINGS ... B-38
B.5.1 Granularity Field (gg) ... B-38
B.5.2 MMX Technology and General-Purpose Register Fields (mmxreg and reg) ... B-38
B.5.3 MMX Instruction Formats and Encodings Table ... B-38
B.6 PROCESSOR EXTENDED STATE INSTRUCTION FORMATS AND ENCODINGS ... B-41
B.7 P6 FAMILY INSTRUCTION FORMATS AND ENCODINGS ... B-41
B.8 SSE INSTRUCTION FORMATS AND ENCODINGS ... B-42
B.9 SSE2 INSTRUCTION FORMATS AND ENCODINGS ... B-47
B.9.1 Granularity Field (gg) ... B-47
B.10 SSE3 FORMATS AND ENCODINGS TABLE ... B-57
B.11 SSSE3 FORMATS AND ENCODING TABLE ... B-58
B.12 AESNI AND PCLMULQDQ INSTRUCTION FORMATS AND ENCODINGS ... B-60
B.13 SPECIAL ENCODINGS FOR 64-BIT MODE ... B-61
B.14 SSE4.1 FORMATS AND ENCODING TABLE ... B-64
B.15 SSE4.2 FORMATS AND ENCODING TABLE ... B-69
B.16 AVX FORMATS AND ENCODING TABLE ... B-70
B.17 FLOATING-POINT INSTRUCTION FORMATS AND ENCODINGS ... B-108
B.18 VMX INSTRUCTIONS ... B-112
B.19 SMX INSTRUCTIONS ... B-113

APPENDIX C
INTEL® C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS
C.1 SIMPLE INTRINSICS ... C-2
C.2 COMPOSITE INTRINSICS ... C-14

PAGE

FIGURES

Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format 2-1
Figure 2-2. Table Interpretation of ModR/M Byte (C8H) 2-4
Figure 2-3. Prefix Ordering in 64-bit Mode 2-8
Figure 2-4. Memory Addressing Without an SIB Byte; REX.X Not Used 2-9
Figure 2-5. Register-Register Addressing (No Memory Operand); REX.X Not Used 2-9
Figure 2-6. Memory Addressing With a SIB Byte 2-10
Figure 2-7. Register Operand Coded in Opcode Byte; REX.X & REX.R Not Used 2-10
Figure 2-8. Instruction Encoding Format with VEX Prefix 2-13
Figure 2-9. VEX bit fields 2-15
Figure 2-10. Intel® AVX-512 Instruction Format and the EVEX Prefix 2-37
Figure 2-11. Bit Field Layout of the EVEX Prefix 2-37
Figure 3-1. Bit Offset for BIT[RAX, 21] 3-12
Figure 3-2. Memory Bit Indexing 3-12
Figure 3-3. ADDSUBPD—Packed Double Precision Floating-Point Add/Subtract 3-27
Figure 3-4. ADDSUBPS—Packed Single Precision Floating-Point Add/Subtract 3-29
Figure 3-5. Memory Layout of BNDMOV to/from Memory 3-99
Figure 3-6. CVTDQ2PD (VEX.256 encoded version) 3-209
Figure 3-7. VCVTPD2DQ (VEX.256 encoded version) 3-215
Figure 3-8. VCVTPD2PS (VEX.256 encoded version) 3-220
Figure 3-9. CVTPS2PD (VEX.256 encoded version) 3-229
Figure 3-10. VCVTTPD2DQ (VEX.256 encoded version) 3-245
Figure 3-11. 64-Byte Data Written to Enqueue Registers 3-289
Figure 3-12. HADDPD—Packed Double Precision Floating-Point Horizontal Add 3-433
Figure 3-13. VHADDPD Operation 3-434
Figure 3-14. HADDPS—Packed Single Precision Floating-Point Horizontal Add 3-437
Figure 3-15. VHADDPS Operation 3-437
Figure 3-16. HSUBPD—Packed Double Precision Floating-Point Horizontal Subtract 3-442
Figure 3-17. VHSUBPD operation 3-443
Figure 3-18. HSUBPS—Packed Single Precision Floating-Point Horizontal Subtract 3-446
Figure 3-19. VHSUBPS Operation 3-446
Figure 3-20. INVPCID Descriptor 3-487
Figure 4-1. Operation of PCMPSTRx and PCMPESTRx 4-5
Figure 4-2. VMOVDDUP Operation 4-48
Figure 4-3. MOVSHDUP Operation 4-111
Figure 4-4. MOVSLDUP Operation 4-114
Figure 4-5. 256-bit VMPSADBW Operation 4-133
Figure 4-6. Operation of the PACKSSDW Instruction Using 64-Bit Operands 4-184
Figure 4-7. 256-bit VPALIGN Instruction Operation 4-217
Figure 4-8. PDEP Example 4-282
Figure 4-9. PEXT Example 4-284
Figure 4-10. 256-bit VPHADDD Instruction Operation 4-295
Figure 4-11. PMADDWD Execution Model Using 64-bit Operands 4-314
Figure 4-12. PMULHUW and PMULHW Instruction Operation Using 64-bit Operands 4-380
Figure 4-13. PMULLU Instruction Operation Using 64-bit Operands 4-392
Figure 4-14. PSADBW Instruction Operation Using 64-bit Operands 4-419
Figure 4-15. PSHUFB with 64-Bit Operands 4-424
Figure 4-16. 256-bit VPSHUFD Instruction Operation 4-427
Figure 4-17. PSLLW, PSLLD, and PSLLQ Instruction Operation Using 64-bit Operand 4-445
Figure 4-18. PSRAW and PSRAD Instruction Operation Using a 64-bit Operand 4-457
Figure 4-19. PSRLW, PSRLD, and PSRLQ Instruction Operation Using 64-bit Operand 4-470
Figure 4-20. PUNPCKHBW Instruction Operation Using 64-bit Operands 4-504
Figure 4-21. 256-bit VPUNPCKHDQ Instruction Operation 4-504
Figure 4-22. PUNPCKLBW Instruction Operation Using 64-bit Operands 4-514
Figure 4-23. 256-bit VPUNPCKLDQ Instruction Operation 4-514
Figure 4-24. Bit Control Fields of Immediate Byte for ROUNDxx Instruction 4-584
Figure 4-25. 256-bit VSHUFPD Operation of Four Pairs of Double Precision Floating-Point Values 4-647
Figure 4-26. 256-bit VSHUFPS Operation of Selection from Input Quadruplet and Pair-wise Interleaved Result 4-652
Figure 4-27. VUNPCKHPS Operation 4-749
Figure 4-28. VUNPCKLPS Operation 4-757
Figure 5-1. VBROADCASTSS Operation (VEX.256 encoded version) 5-14
Figure 5-2. VBROADCASTSS Operation (VEX.128-bit version) 5-14
Figure 5-3. VBROADCASTSD Operation (VEX.256-bit version) 5-14

PAGE

Figure 5-4. VBROADCASTF128 Operation (VEX.256-bit version) 5-14
Figure 5-5. VBROADCASTF64X4 Operation (512-bit version with writemask all 1s) 5-15
Figure 5-6. VCVTPH2PS (128-bit Version) 5-55
Figure 5-7. VCVTPS2PH (128-bit Version) 5-68
Figure 5-8. 64-bit Super Block of SAD Operation in VDBPSADBW 5-155
Figure 5-9. VFIXUPIMMPD Immediate Control Description 5-193
Figure 5-10. VFIXUPIMMPS Immediate Control Description 5-197
Figure 5-11. VFIXUPIMMSD Immediate Control Description 5-201
Figure 5-12. VFIXUPIMMSS Immediate Control Description 5-205
Figure 5-13. Imm8 Byte Specifier of Special Case Floating-Point Values for VFPCLASSPD/SD/PS/SS 5-345
Figure 5-14. VGETEXPPS Functionality On Normal Input values. 5-380
Figure 5-15. Imm8 Controls for VGETMANTPD/SD/PS/SS. 5-389
Figure 5-16. VPBROADCASTD Operation (VEX.256 encoded version) 5-444
Figure 5-17. VPBROADCASTD Operation (128-bit version) 5-444
Figure 5-18. VPBROADCASTQ Operation (256-bit version) 5-445
Figure 5-19. VBROADCASTI128 Operation (256-bit version) 5-445
Figure 5-20. VBROADCASTI256 Operation (512-bit version) 5-445
Figure 5-21. VPERM2F128 Operation 5-494
Figure 5-22. VPERM2I128 Operation 5-496
Figure 5-23. VPERMILPD Operation 5-512
Figure 5-24. VPERMILPD Shuffle Control 5-512
Figure 5-25. VPERMILPS Operation 5-518
Figure 5-26. VPERMILPS Shuffle Control 5-518
Figure 5-27. Imm8 Controls for VRANGEPD/SD/PS/SS. 5-661
Figure 5-28. Imm8 Controls for VREDUCEPD/SD/PS/SS. 5-686
Figure 5-29. Imm8 Controls for VRNDSCALEPD/SD/PS/SS 5-700
Figure 8-1. Register Source-Block Dot Product of Two Signed Word Operands With Doubleword Accumulation 8-18
Figure A-1. ModR/M Byte nnn Field (Bits 5, 4, and 3). A-17
Figure B-1. General Machine Instruction Format B-1
Figure B-2. Hybrid Notation of VEX-Encoded Key Instruction Bytes B-70

PAGE

TABLES

Table 2-1. 16-Bit Addressing Forms with the ModR/M Byte 2-5
Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte 2-6
Table 2-3. 32-Bit Addressing Forms with the SIB Byte 2-7
Table 2-4. REX Prefix Fields [BITS: 0100WRXB] 2-9
Table 2-5. Special Cases of REX Encodings 2-10
Table 2-6. Direct Memory Offset Form of MOV 2-11
Table 2-7. RIP-Relative Addressing. 2-12
Table 2-8. VEX.vvvv to Register Name Mapping 2-17
Table 2-9. Instructions with a VEX.vvvv Destination 2-17
Table 2-10. VEX.m-mmmm Interpretation 2-18
Table 2-11. VEX.L Interpretation 2-19
Table 2-12. VEX.pp Interpretation. 2-19
Table 2-13. 32-Bit VSIB Addressing Forms of the SIB Byte 2-21
Table 2-14. Exception Class Description 2-22
Table 2-15. Instructions in Each Exception Class 2-23
Table 2-16. #UD Exception and VEX.W=1 Encoding 2-24
Table 2-17. #UD Exception and VEX.L Field Encoding. 2-25
Table 2-18. Type 1 Class Exception Conditions. 2-26
Table 2-19. Type 2 Class Exception Conditions. 2-27
Table 2-20. Type 3 Class Exception Conditions. 2-28
Table 2-21. Type 4 Class Exception Conditions. 2-29
Table 2-22. Type 5 Class Exception Conditions. 2-30
Table 2-23. Type 6 Class Exception Conditions. 2-31
Table 2-24. Type 7 Class Exception Conditions. 2-32
Table 2-25. Type 8 Class Exception Conditions. 2-32
Table 2-26. Type 11 Class Exception Conditions 2-33
Table 2-27. Type 12 Class Exception Conditions 2-34
Table 2-28. VEX-Encoded GPR Instructions 2-35
Table 2-29. Type 13 Class Exception Conditions 2-35
Table 2-30. Exceptions Type 14 Instructions 2-36
Table 2-31. Type 14 Class Exception Conditions 2-36
Table 2-32. EVEX Prefix Bit Field Functional Grouping. 2-38
Table 2-33. 32-Register Support in 64-bit Mode Using EVEX with Embedded REX Bits 2-39
Table 2-34. EVEX Encoding Register Specifiers in 32-bit Mode. 2-39
Table 2-35. Opmask Register Specifier Encoding 2-40
Table 2-36. Compressed Displacement (DISP8*N) Affected by Embedded Broadcast 2-41
Table 2-37. EVEX DISP8*N for Instructions Not Affected by Embedded Broadcast 2-41
Table 2-38. EVEX Embedded Broadcast/Rounding/SAE and Vector Length on Vector Instructions 2-43
Table 2-39. OS XSAVE Enabling Requirements of Instruction Categories 2-43
Table 2-40. Opcode Independent, State Dependent EVEX Bit Fields 2-43
Table 2-41. #UD Conditions of Operand-Encoding EVEX Prefix Bit Fields 2-44
Table 2-42. #UD Conditions of Opmask Related Encoding Field. 2-44
Table 2-43. #UD Conditions Dependent on EVEX.b Context. 2-45
Table 2-44. EVEX-Encoded Instruction Exception Class Summary 2-46
Table 2-45. EVEX Instructions in Each Exception Class 2-46
Table 2-46. Type E1 Class Exception Conditions. 2-49
Table 2-47. Type E1NF Class Exception Conditions. 2-50
Table 2-48. Type E2 Class Exception Conditions. 2-51
Table 2-49. Type E3 Class Exception Conditions. 2-52
Table 2-50. Type E3NF Class Exception Conditions. 2-53
Table 2-51. Type E4 Class Exception Conditions. 2-54
Table 2-52. Type E4NF Class Exception Conditions. 2-55
Table 2-53. Type E5 Class Exception Conditions. 2-56
Table 2-54. Type E5NF Class Exception Conditions. 2-57
Table 2-55. Type E6 Class Exception Conditions. 2-58
Table 2-56. Type E6NF Class Exception Conditions. 2-59
Table 2-57. Type E7NM Class Exception Conditions 2-60

PAGE

Table 2-58. Type E9 Class Exception Conditions 2-60
Table 2-59. Type E9NF Class Exception Conditions 2-62
Table 2-60. Type E10 Class Exception Conditions 2-63
Table 2-61. Type E10NF Class Exception Conditions 2-64
Table 2-62. Type E11 Class Exception Conditions 2-65
Table 2-63. Type E12 Class Exception Conditions 2-66
Table 2-64. Type E12NP Class Exception Conditions 2-67
Table 2-65. TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg) 2-68
Table 2-66. TYPE K21 Exception Definition (VEX-Encoded OpMask Instructions Addressing Memory) 2-68
Table 2-67. Intel® AMX Exception Classes 2-69
Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro 3-2
Table 3-2. Range of Bit Positions Specified by Bit Offset Operands 3-12
Table 3-3. Standard and Non-Standard Data Types 3-14
Table 3-4. Intel 64® and IA-32 General Exceptions 3-15
Table 3-5. x87 FPU Floating-Point Exceptions 3-16
Table 3-6. SIMD Floating-Point Exceptions 3-17
Table 3-7. Decision Table for CLI Results 3-148
Table 3-8. Comparison Predicate for CMPPD and CMPPS Instructions 3-169
Table 3-9. Pseudo-Op and CMPPD Implementation 3-170
Table 3-10. Pseudo-Op and VCMPPD Implementation 3-171
Table 3-11. Pseudo-Op and CMPPS Implementation 3-176
Table 3-12. Pseudo-Op and VCMPPS Implementation 3-177
Table 3-13. Pseudo-Op and CMPSD Implementation 3-186
Table 3-14. Pseudo-Op and VCMPSD Implementation 3-186
Table 3-15. Pseudo-Op and CMPSS Implementation 3-190
Table 3-16. Pseudo-Op and VCMPSS Implementation 3-190
Table 3-17. DIV Action 3-264
Table 3-18. Results Obtained from F2XM1 3-307
Table 3-19. Results Obtained from FABS 3-309
Table 3-20. FADD/FADDP/FIADD Results 3-312
Table 3-21. FBSTP Results 3-316
Table 3-22. FCHS Results 3-318
Table 3-23. FCOM/FCOMP/FCOMPP Results 3-324
Table 3-24. FCOMI/FCOMIP/ FUCOMI/FUCOMIP Results 3-327
Table 3-25. FCOS Results 3-330
Table 3-26. FDIV/FDIVP/FIDIV Results 3-334
Table 3-27. FDIVR/FDIVRP/FIDIVR Results 3-337
Table 3-28. FICOM/FICOMP Results 3-340
Table 3-29. FIST/FISTP Results 3-347
Table 3-30. FISTTP Results 3-350
Table 3-31. FMUL/FMULP/FIMUL Results 3-361
Table 3-32. FPATAN Results 3-364
Table 3-33. FPREM Results 3-366
Table 3-34. FPREM1 Results 3-368
Table 3-35. FPTAN Results 3-370
Table 3-36. FSCALE Results 3-378
Table 3-37. FSIN Results 3-380
Table 3-38. FSINCOS Results 3-382
Table 3-39. FSQRT Results 3-384
Table 3-40. FSUB/FSUBP/FISUB Results 3-395
Table 3-41. FSUBR/FSUBRP/FISUBR Results 3-398
Table 3-42. FTST Results 3-400
Table 3-43. FUCOM/FUCOMP/FUCOMPP Results 3-402
Table 3-44. FXAM Results 3-405
Table 3-45. Non-64-Bit-Mode Layout of FXSAVE and FXRSTOR Memory Region 3-412
Table 3-46. Field Definitions 3-413
Table 3-47. Recreating FSAVE Format 3-415
Table 3-48. Layout of the 64-Bit Mode FXSAVE64 Map (Requires REX.W = 1) 3-416
Table 3-49. Layout of the 64-Bit Mode FXSAVE Map (REX.W = 0) 3-417

PAGE

Table 3-50. FYL2X Results....3-422
Table 3-51. FYL2XP1 Results....3-424
Table 3-52. Inverse Byte Listings....3-427
Table 3-53. IDIV Results....3-448
Table 3-54. Decision Table....3-468
Table 3-55. Segment and Gate Types....3-534
Table 3-56. Memory Area Layout....3-544
Table 3-57. Non-64-bit Mode LEA Operation with Address and Operand Size Attributes....3-547
Table 3-58. 64-bit Mode LEA Operation with Address and Operand Size Attributes....3-547
Table 3-59. Segment and Gate Descriptor Types....3-574
Table 4-1. Source Data Format....4-1
Table 4-2. Aggregation Operation....4-2
Table 4-3. Aggregation Operation....4-2
Table 4-4. Polarity....4-3
Table 4-5. Output Selection....4-3
Table 4-6. Output Selection....4-3
Table 4-7. Comparison Result for Each Element Pair BoolRes[i,j]....4-3
Table 4-8. Summary of Imm8 Control Byte....4-4
Table 4-9. MUL Results....4-140
Table 4-10. MWAIT Extension Register (ECX)....4-155
Table 4-11. MWAIT Hints Register (EAX)....4-155
Table 4-12. Recommended Multi-Byte Sequence of NOP Instruction....4-160
Table 4-13. Bind Structure Format....4-238
Table 4-14. PCLMULQDQ Quadword Selection of Immediate Byte....4-243
Table 4-15. Pseudo-Op and PCLMULQDQ Implementation....4-243
Table 4-16. MKTME_KEY_PROGRAM_STRUCT Format....4-271
Table 4-17. TSE_KEY_PROGRAM_STRUCT Format....4-273
Table 4-18. TSE_KEY_PROGRAM_WRAPPED Control Input....4-274
Table 4-19. Bind Structure Format....4-275
Table 4-20. Effect of POPF/POPFD on the EFLAGS Register....4-408
Table 4-21. Rounding Modes and Encoding of Rounding Control (RC) Field....4-584
Table 4-22. Decision Table for STI Results....4-674
Table 4-23. TPAUSE Input Register Bit Definitions....4-729
Table 4-24. UMWAIT Input Register Bit Definitions....4-742
Table 5-1. Lower 8 columns of the 16x16 Map of VPTERNLOG Boolean Logic Operations....5-2
Table 5-2. Upper 8 columns of the 16x16 Map of VPTERNLOG Boolean Logic Operations....5-3
Table 5-3. Immediate Byte Encoding for 16-bit Floating-Point Conversion Instructions....5-69
Table 5-4. NaN Propagation Priorities....5-160
Table 5-5. VF[,N]MADD[132,213,231]PH Notation for Operands....5-215
Table 5-6. VF[,N]MADD[132,213,231]SH Notation for Operands....5-229
Table 5-7. VFMADDSUB[132,213,231]PH Notation for Odd and Even Elements....5-244
Table 5-8. VF[,N]MSUB[132,213,231]PH Notation for Operands....5-264
Table 5-9. VF[,N]MSUB[132,213,231]SH Notation for Operands....5-279
Table 5-10. VFMSUBADD[132,213,231]PH Notation for Odd and Even Elements....5-293
Table 5-11. Classifier Operations for VFPCLASSPD/SD/PS/SS....5-345
Table 5-12. Classifier Operations for VFPCLASSPH/VFPCLASSSH....5-348
Table 5-13. VGETEXPPD/SD Special Cases....5-372
Table 5-14. VGETEXPPH/VGETEXPSH Special Cases....5-376
Table 5-15. VGETEXPPS/SS Special Cases....5-380
Table 5-16. GetMant() Special Float Values Behavior....5-390
Table 5-17. imm8 Controls for VGETMANTPH/VGETMANTSH....5-393
Table 5-18. GetMant() Special Float Values Behavior....5-394
Table 5-19. Pseudo-Op and VPCMP* Implementation....5-457
Table 5-20. Examples of VPTERNLOGD/Q Imm8 Boolean Function and Input Index Values....5-652
Table 5-21. Signaling of Comparison Operation of One or More NaN Input Values and Effect of Imm8[3:2]....5-662
Table 5-22. Comparison Result for Opposite-Signed Zero Cases for MIN, MIN_ABS, and MAX, MAX_ABS....5-662
Table 5-23. Comparison Result of Equal-Magnitude Input Cases for MIN_ABS and MAX_ABS, (|a| = |b|, a>0, b<0)....5-662
Table 5-24. VRCP14PD/VRCP14SD Special Cases....5-674
Table 5-25. VRCP14PS/VRCP14SS Special Cases....5-676

PAGE

Table 5-26. VRCPPH/VRCPSH Special Cases 5-682
Table 5-27. VREDUCEPD/SD/PS/SS Special Cases 5-686
Table 5-28. VREDUCEPH/VREDUCESH Special Cases 5-689
Table 5-29. VRNDSCALEPD/SD/PS/SS Special Cases. 5-700
Table 5-30. Imm8 Controls for VRNDSCALEPH/VRNDSCALESH 5-703
Table 5-31. VRNDSCALEPH/VRNDSCALESH Special Cases 5-703
Table 5-32. VRSQRT14PD Special Cases 5-715
Table 5-33. VRSQRT14PS Special Cases. 5-717
Table 5-34. VRSQRT14SD Special Cases. 5-719
Table 5-35. VRSQRT14SS Special Cases. 5-721
Table 5-36. VRSQRTPH/VRSQRTSH Special Cases 5-722
Table 5-37. VSCALEFPD/SD/PS/SS Special Cases. 5-726
Table 5-38. Additional VSCALEFPD/SD Special Cases 5-726
Table 5-39. VSCALEFPH/VSCALEFSH Special Cases 5-728
Table 5-40. Additional VSCALEFPH/VSCALEFSH Special Cases 5-728
Table 5-41. Additional VSCALEFPS/SS Special Cases 5-731
Table 7-1. Layout of IA32_FEATURE_CONTROL 7-1
Table 7-2. GETSEC Leaf Functions 7-3
Table 7-3. GETSEC Capability Result Encoding (EBX = 0) 7-7
Table 7-4. Register State Initialization After GETSEC[ENTERACCS] 7-12
Table 7-5. IA32_MISC_ENABLE MSR Initialization by ENTERACCS and SENTER 7-13
Table 7-6. Register State Initialization After GETSEC[SENTER] and GETSEC[WAKEUP] 7-25
Table 7-7. SMX Reporting Parameters Format. 7-34
Table 7-8. TXT Feature Extensions Flags 7-34
Table 7-9. External Memory Types Using Parameter 3 7-35
Table 7-10. Default Parameter Values 7-36
Table 7-11. Supported Actions for GETSEC[SMCTRL(0)] 7-38
Table 7-12. RLP MVMM JOIN Data Structure 7-41
Table 8-1. Special Values Behavior. 8-9
Table 8-2. Special Values Behavior. 8-11
Table 8-3. VRCP28PD Special Cases 8-21
Table 8-4. VRCP28SD Special Cases 8-23
Table 8-5. VRCP28PS Special Cases. 8-25
Table 8-6. VRCP28SS Special Cases. 8-27
Table 8-7. VRSQRT28PD Special Cases 8-29
Table 8-8. VRSQRT28SD Special Cases. 8-31
Table 8-9. VRSQRT28PS Special Cases. 8-33
Table 8-10. VRSQRT28SS Special Cases. 8-35
Table A-1. Superscripts Utilized in Opcode Tables. A-6
Table A-2. One-byte Opcode Map: (00H — F7H) * A-7
Table A-3. Two-byte Opcode Map: 00H — 77H (First Byte is 0FH) * A-9
Table A-4. Three-byte Opcode Map: 00H — F7H (First Two Bytes are 0F 38H) * A-13
Table A-5. Three-byte Opcode Map: 00H — F7H (First two bytes are 0F 3AH) * A-15
Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number * A-18
Table A-7. D8 Opcode Map When ModR/M Byte is Within 00H to BFH * A-21
Table A-8. D8 Opcode Map When ModR/M Byte is Outside 00H to BFH * A-22
Table A-9. D9 Opcode Map When ModR/M Byte is Within 00H to BFH * A-22
Table A-10. D9 Opcode Map When ModR/M Byte is Outside 00H to BFH * A-23
Table A-11. DA Opcode Map When ModR/M Byte is Within 00H to BFH * A-23
Table A-12. DA Opcode Map When ModR/M Byte is Outside 00H to BFH * A-24
Table A-13. DB Opcode Map When ModR/M Byte is Within 00H to BFH * A-24
Table A-14. DB Opcode Map When ModR/M Byte is Outside 00H to BFH * A-25
Table A-15. DC Opcode Map When ModR/M Byte is Within 00H to BFH *. A-25
Table A-16. DC Opcode Map When ModR/M Byte is Outside 00H to BFH * A-26
Table A-17. DD Opcode Map When ModR/M Byte is Within 00H to BFH * A-26
Table A-18. DD Opcode Map When ModR/M Byte is Outside 00H to BFH * A-27
Table A-19. DE Opcode Map When ModR/M Byte is Within 00H to BFH *. A-27
Table A-20. DE Opcode Map When ModR/M Byte is Outside 00H to BFH * A-28
Table A-21. DF Opcode Map When ModR/M Byte is Within 00H to BFH *. A-28

PAGE

Table A-22. DF Opcode Map When ModR/M Byte is Outside 00H to BFH * A-29
Table B-1. Special Fields Within Instruction Encodings B-2
Table B-2. Encoding of reg Field When w Field is Not Present in Instruction B-3
Table B-3. Encoding of reg Field When w Field is Present in Instruction B-3
Table B-4. Encoding of reg Field When w Field is Not Present in Instruction B-4
Table B-5. Encoding of reg Field When w Field is Present in Instruction B-4
Table B-6. Encoding of Operand Size (w) Bit B-4
Table B-7. Encoding of Sign-Extend (s) Bit B-5
Table B-8. Encoding of the Segment Register (sreg) Field B-5
Table B-9. Encoding of Special-Purpose Register (eee) Field B-5
Table B-10. Encoding of Conditional Test (tttn) Field B-6
Table B-11. Encoding of Operation Direction (d) Bit B-6
Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes B-7
Table B-12. Notes on Instruction Encoding B-7
Table B-14. Special Symbols B-18
Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode B-18
Table B-16. Pentium® Processor Family Instruction Formats and Encodings, Non-64-Bit Modes B-37
Table B-17. Pentium® Processor Family Instruction Formats and Encodings, 64-Bit Mode B-37
Table B-18. Encoding of Granularity of Data Field (gg) B-38
Table B-19. MMX Instruction Formats and Encodings B-38
Table B-20. Formats and Encodings of XSAVE/XRSTOR/XGETBV/XSETBV Instructions B-41
Table B-21. Formats and Encodings of P6 Family Instructions B-41
Table B-22. Formats and Encodings of SSE Floating-Point Instructions B-42
Table B-23. Formats and Encodings of SSE Integer Instructions B-46
Table B-25. Encoding of Granularity of Data Field (gg) B-47
Table B-24. Format and Encoding of SSE Cacheability & Memory Ordering Instructions B-47
Table B-26. Formats and Encodings of SSE2 Floating-Point Instructions B-48
Table B-27. Formats and Encodings of SSE2 Integer Instructions B-52
Table B-28. Format and Encoding of SSE2 Cacheability Instructions B-56
Table B-29. Formats and Encodings of SSE3 Floating-Point Instructions B-57
Table B-30. Formats and Encodings for SSE3 Event Management Instructions B-57
Table B-31. Formats and Encodings for SSE3 Integer and Move Instructions B-57
Table B-32. Formats and Encodings for SSSE3 Instructions B-58
Table B-33. Formats and Encodings of AESNI and PCLMULQDQ Instructions B-61
Table B-34. Special Case Instructions Promoted Using REX.W B-61
Table B-35. Encodings of SSE4.1 instructions B-64
Table B-36. Encodings of SSE4.2 instructions B-69
Table B-37. Encodings of AVX Instructions B-71
Table B-38. General Floating-Point Instruction Formats B-108
Table B-39. Floating-Point Instruction Formats and Encodings B-108
Table B-40. Encodings for VMX Instructions B-112
Table B-41. Encodings for SMX Instructions B-113
Table C-1. Simple Intrinsics C-2
Table C-2. Composite Intrinsics C-14

CHAPTER 1
ABOUT THIS MANUAL

The Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D: Instruction Set Reference (order numbers 253666, 253667, 326018, and 334569), is part of a set that describes the architecture and programming environment of all Intel 64 and IA-32 architecture processors. Other volumes in this set are:

- The Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture (Order Number 253665).
- The Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C, & 3D: System Programming Guide (order numbers 253668, 253669, 326019, and 332831).
- The Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers (order number 335592).

The Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, describes the basic architecture and programming environment of Intel 64 and IA-32 processors. The Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D, describes the instruction set of the processor and the opcode structure. These volumes apply to application programmers and to programmers who write operating systems or executives. The Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C, & 3D, describes the operating-system support environment of Intel 64 and IA-32 processors. These volumes target operating-system and BIOS designers. In addition, the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B, addresses the programming environment for classes of software that host operating systems. The Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4, describes the model-specific registers of Intel 64 and IA-32 processors.

1.1 OVERVIEW OF VOLUME 2A, 2B, 2C, AND 2D: INSTRUCTION SET REFERENCE

A description of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D, content follows:

Chapter 1 — About This Manual. Gives an overview of all volumes of the Intel® 64 and IA-32 Architectures Software Developer's Manual, with chapter-specific details for the current volume.

Chapter 2 — Instruction Format. Describes the machine-level instruction format used for all IA-32 instructions and gives the allowable encodings of prefixes, the operand-identifier byte (ModR/M byte), the addressing-mode specifier byte (SIB byte), and the displacement and immediate bytes.

Chapter 3 — Instruction Set Reference, A-L. Describes Intel 64 and IA-32 instructions in detail, including an algorithmic description of operations, the effect on flags, the effect of operand- and address-size attributes, and the exceptions that may be generated. The instructions are arranged in alphabetical order. General-purpose, x87 FPU, Intel MMX™ technology, SSE/SSE2/SSE3/SSSE3/SSE4 extensions, and system instructions are included.

Chapter 4 — Instruction Set Reference, M-U. Continues the description of Intel 64 and IA-32 instructions started in Chapter 3. It starts Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B.

Chapter 5 — Instruction Set Reference, V. Continues the description of Intel 64 and IA-32 instructions started in chapters 3 and 4. This chapter starts Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C.

Chapter 6 — Instruction Set Reference, W-Z. Continues the description of Intel 64 and IA-32 instructions started in chapters 3, 4, and 5. It provides the balance of the alphabetized list of instructions and starts Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D.

Chapter 7 — Safer Mode Extensions Reference. Describes the safer mode extensions (SMX). SMX is intended for a system executive to support launching a measured environment in a platform where the identity of the software controlling the platform hardware can be measured for the purpose of making trust decisions.

Chapter 8— Instruction Set Reference Unique to Intel® Xeon Phi™ Processors. Describes the instruction set that is unique to Intel® Xeon Phi™ processors based on the Knights Landing and Knights Mill microarchitectures. The set is not supported in any other Intel processors.

Appendix A — Opcode Map. Gives an opcode map for the IA-32 instruction set.

Appendix B — Instruction Formats and Encodings. Gives the binary encoding of each form of each IA-32 instruction.

Appendix C — Intel® C/C++ Compiler Intrinsics and Functional Equivalents. Lists the Intel® C/C++ compiler intrinsics and their assembly code equivalents for each of the IA-32 MMX and SSE/SSE2/SSE3 instructions.

CHAPTER 2
INSTRUCTION FORMAT

This chapter describes the instruction format for all Intel 64 and IA-32 processors. The instruction format for protected mode, real-address mode and virtual-8086 mode is described in Section 2.1. Increments provided for IA-32e mode and its sub-modes are described in Section 2.2.

2.1 INSTRUCTION FORMAT FOR PROTECTED MODE, REAL-ADDRESS MODE, AND VIRTUAL-8086 MODE

The Intel 64 and IA-32 architectures instruction encodings are subsets of the format shown in Figure 2-1. Instructions consist of optional instruction prefixes (in any order), primary opcode bytes (up to three bytes), an addressing-form specifier (if required) consisting of the ModR/M byte and sometimes the SIB (Scale-Index-Base) byte, a displacement (if required), and an immediate data field (if required).

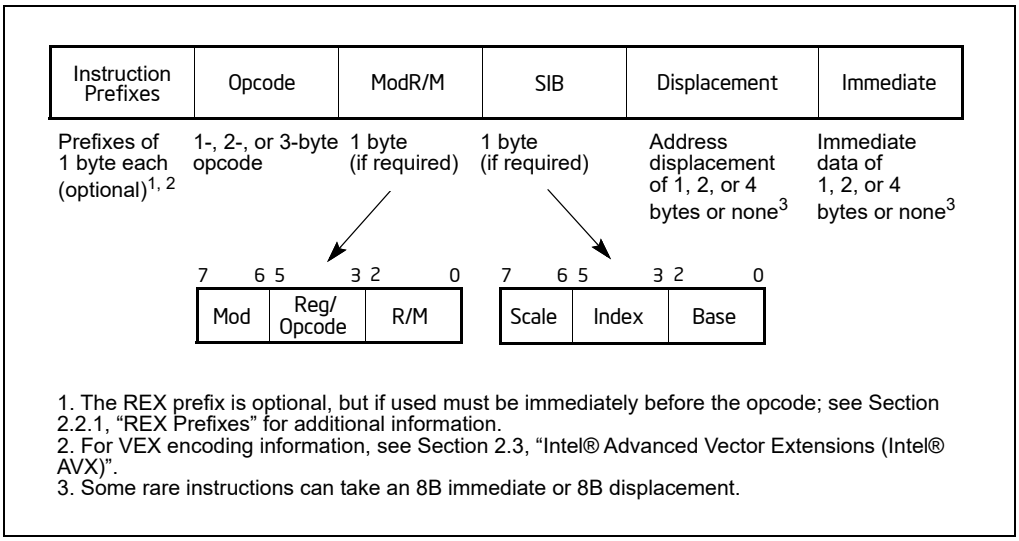


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

2.1.1 Instruction Prefixes

Instruction prefixes are divided into four groups, each with a set of allowable prefix codes. For each instruction, it is only useful to include up to one prefix code from each of the four groups (Groups 1, 2, 3, 4). Groups 1 through 4 may be placed in any order relative to each other.

- Group 1
  - — Lock and repeat prefixes:
    - LOCK prefix is encoded using F0H.
    - REPNE/REPNZ prefix is encoded using F2H. Repeat-Not-Zero prefix applies only to string and input/output instructions. (F2H is also used as a mandatory prefix for some instructions.)
    - REP or REPE/REPZ is encoded using F3H. The repeat prefix applies only to string and input/output instructions. (F3H is also used as a mandatory prefix for some instructions.)

  - BND prefix is encoded using F2H if the following conditions are true:
    - CPUID.07H.00H:EBX.MPX[14] is set.
    - BNDCFGU.EN and/or IA32_BNDCFGS.EN is set.
    - When the F2 prefix precedes a near CALL, a near RET, a near JMP, a short Jcc, or a near Jcc instruction (see Appendix E, "Intel® Memory Protection Extensions," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1).
- Group 2
  - Segment override prefixes:
    - 2EH—CS segment override (use with any branch instruction is reserved).
    - 36H—SS segment override prefix (use with any branch instruction is reserved).
    - 3EH—DS segment override prefix (use with any branch instruction is reserved).
    - 26H—ES segment override prefix (use with any branch instruction is reserved).
    - 64H—FS segment override prefix (use with any branch instruction is reserved).
    - 65H—GS segment override prefix (use with any branch instruction is reserved).
  - Branch hints[1]:
    - 2EH—Branch not taken (used only with J*cc* instructions).
    - 3EH—Branch taken (used only with J*cc* instructions).
- Group 3
    - Operand-size override prefix is encoded using 66H (66H is also used as a mandatory prefix for some instructions).
- Group 4
    - 67H—Address-size override prefix.

The LOCK prefix (F0H) forces an operation that ensures exclusive use of shared memory in a multiprocessor environment. See "LOCK—Assert LOCK# Signal Prefix" in Chapter 3, "Instruction Set Reference, A-L," for a description of this prefix.

Repeat prefixes (F2H, F3H) cause an instruction to be repeated for each element of a string. Use these prefixes only with string and I/O instructions (MOVS, CMPS, SCAS, LODS, STOS, INS, and OUTS). Use of repeat prefixes and/or undefined opcodes with other Intel 64 or IA-32 instructions is reserved; such use may cause unpredictable behavior.

Some instructions may use F2H or F3H as a mandatory prefix to express distinct functionality.

Branch hint prefixes (2EH, 3EH) allow a program to give a hint to the processor about the most likely code path for a branch when used on conditional branch instructions (*Jcc*).

The operand-size override prefix allows a program to switch between 16- and 32-bit operand sizes. Either size can be the default; use of the prefix selects the non-default size.

Some SSE2/SSE3/SSSE3/SSE4 instructions and instructions using a three-byte sequence of primary opcode bytes may use 66H as a mandatory prefix to express distinct functionality.

Other use of the 66H prefix is reserved; such use may cause unpredictable behavior.

The address-size override prefix (67H) allows programs to switch between 16- and 32-bit addressing. Either size can be the default; the prefix selects the non-default size. Using this prefix and/or other undefined opcodes when operands for the instruction do not reside in memory is reserved; such use may cause unpredictable behavior.

2.1.2 Opcodes

A primary opcode can be 1, 2, or 3 bytes in length. An additional 3-bit opcode field is sometimes encoded in the ModR/M byte. Smaller fields can be defined within the primary opcode. Such fields define the direction of operation,

1. Microarchitectural behavior varies; refer to the Intel® 64 and IA-32 Architectures Optimization Reference Manual.

size of displacements, register encoding, condition codes, or sign extension. Encoding fields used by an opcode vary depending on the class of operation.

Two-byte opcode formats for general-purpose and SIMD instructions consist of one of the following:

- An escape opcode byte 0FH as the primary opcode and a second opcode byte.
- A mandatory prefix (66H, F2H, or F3H), an escape opcode byte, and a second opcode byte (same as previous bullet).

For example, CVTDQ2PD consists of the following sequence: F3 0F E6. The first byte is a mandatory prefix (it is not considered as a repeat prefix).

Three-byte opcode formats for general-purpose and SIMD instructions consist of one of the following:

- An escape opcode byte 0FH as the primary opcode, plus two additional opcode bytes.
- A mandatory prefix (66H, F2H, or F3H), an escape opcode byte, plus two additional opcode bytes (same as previous bullet).

For example, PHADDW for XMM registers consists of the following sequence: 66 0F 38 01. The first byte is the mandatory prefix.

Valid opcode expressions are defined in Appendix A and Appendix B.

2.1.3 ModR/M and SIB Bytes

Many instructions that refer to an operand in memory have an addressing-form specifier byte (called the ModR/M byte) following the primary opcode. The ModR/M byte contains three fields of information:

- The *mod* field combines with the r/m field to form 32 possible values: eight registers and 24 addressing modes.
- The *reg/opcode* field specifies either a register number or three more bits of opcode information. The purpose of the reg/opcode field is specified in the primary opcode.
- The *r/m* field can specify a register as an operand or it can be combined with the mod field to encode an addressing mode. Sometimes, certain combinations of the *mod* field and the *r/m* field are used to express opcode information for some instructions.

Certain encodings of the ModR/M byte require a second addressing byte (the SIB byte). The base-plus-index and scale-plus-index forms of 32-bit addressing require the SIB byte. The SIB byte includes the following fields:

- The *scale* field specifies the scale factor.
- The *index* field specifies the register number of the index register.
- The *base* field specifies the register number of the base register.

See Section 2.1.5 for the encodings of the ModR/M and SIB bytes.

2.1.4 Displacement and Immediate Bytes

Some addressing forms include a displacement immediately following the ModR/M byte (or the SIB byte if one is present). If a displacement is required, it can be 1, 2, or 4 bytes.

If an instruction specifies an immediate operand, the operand always follows any displacement bytes. An immediate operand can be 1, 2 or 4 bytes.

2.1.5 Addressing-Mode Encoding of ModR/M and SIB Bytes

The values and corresponding addressing forms of the ModR/M and SIB bytes are shown in Table 2-1 through Table 2-3: 16-bit addressing forms specified by the ModR/M byte are in Table 2-1 and 32-bit addressing forms are in Table 2-2. Table 2-3 shows 32-bit addressing forms specified by the SIB byte. In cases where the reg/opcode field in the ModR/M byte represents an extended opcode, valid encodings are shown in Appendix B.

In Table 2-1 and Table 2-2, the Effective Address column lists 32 effective addresses that can be assigned to the first operand of an instruction by using the Mod and R/M fields of the ModR/M byte. The first 24 options provide ways of specifying a memory location; the last eight (Mod = 11B) provide ways of specifying general-purpose, MMX technology and XMM registers.

The Mod and R/M columns in Table 2-1 and Table 2-2 give the binary encodings of the Mod and R/M fields required to obtain the effective address listed in the first column. For example: see the row indicated by Mod = 11B, R/M = 000B. The row identifies the general-purpose registers EAX, AX or AL; MMX technology register MM0; or XMM register XMM0. The register used is determined by the opcode byte and the operand-size attribute.

Now look at the seventh row in either table (labeled "REG ="). This row specifies the use of the 3-bit Reg/Opcode field when the field is used to give the location of a second operand. The second operand must be a general-purpose, MMX technology, or XMM register. Rows one through five list the registers that may correspond to the value in the table. Again, the register used is determined by the opcode byte along with the operand-size attribute.

If the instruction does not require a second operand, then the Reg/Opcode field may be used as an opcode extension. This use is represented by the sixth row in the tables (labeled "/digit (Opcode)"). Note that values in row six are represented in decimal form.

The body of Table 2-1 and Table 2-2 (under the label "Value of ModR/M Byte (in Hexadecimal)") contains a 32 by 8 array that presents all of 256 values of the ModR/M byte (in hexadecimal). Bits 3, 4, and 5 are specified by the column of the table in which a byte resides. The row specifies bits 0, 1, and 2; and bits 6 and 7. The figure below demonstrates interpretation of one table value.

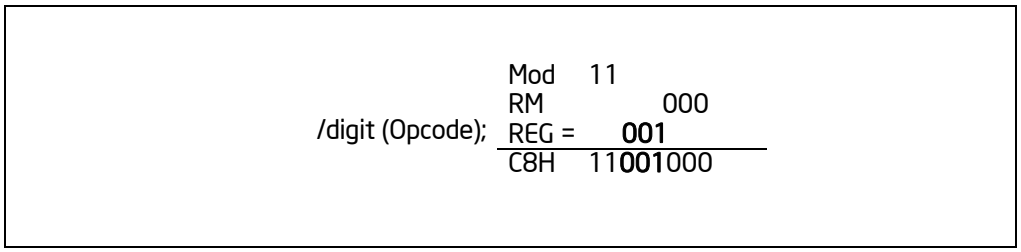


Figure 2-2. Table Interpretation of ModR/M Byte (C8H)

Table 2-1. 16-Bit Addressing Forms with the ModR/M Byte

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP[1] EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[BX+SI] [BX+DI] [BP+SI] [BP+DI] [SI] [DI] disp16[2] [BX]	00	000 001 010 011 100 101 110 111	00 01 02 03 04 05 06 07	08 09 0A 0B 0C 0D 0E 0F	10 11 12 13 14 15 16 17	18 19 1A 1B 1C 1D 1E 1F	20 21 22 23 24 25 26 27	28 29 2A 2B 2C 2D 2E 2F	30 31 32 33 34 35 36 37	38 39 3A 3B 3C 3D 3E 3F
[BX+SI]+disp8[3] [BX+DI]+disp8 [BP+SI]+disp8 [BP+DI]+disp8 [SI]+disp8 [DI]+disp8 [BP]+disp8 [BX]+disp8	01	000 001 010 011 100 101 110 111	40 41 42 43 44 45 46 47	48 49 4A 4B 4C 4D 4E 4F	50 51 52 53 54 55 56 57	58 59 5A 5B 5C 5D 5E 5F	60 61 62 63 64 65 66 67	68 69 6A 6B 6C 6D 6E 6F	70 71 72 73 74 75 76 77	78 79 7A 7B 7C 7D 7E 7F
[BX+SI]+disp16 [BX+DI]+disp16 [BP+SI]+disp16 [BP+DI]+disp16 [SI]+disp16 [DI]+disp16 [BP]+disp16 [BX]+disp16	10	000 001 010 011 100 101 110 111	80 81 82 83 84 85 86 87	88 89 8A 8B 8C 8D 8E 8F	90 91 92 93 94 95 96 97	98 99 9A 9B 9C 9D 9E 9F	A0 A1 A2 A3 A4 A5 A6 A7	A8 A9 AA AB AC AD AE AF	B0 B1 B2 B3 B4 B5 B6 B7	B8 B9 BA BB BC BD BE BF
EAX/AX/AL/MM0/XMM0 ECX/CX/CL/MM1/XMM1 EDX/DX/DL/MM2/XMM2 EBX/BX/BL/MM3/XMM3 ESP/SP/AHMM4/XMM4 EBP/BP/CH/MM5/XMM5 ESI/SI/DH/MM6/XMM6 EDI/DI/BH/MM7/XMM7	11	000 001 010 011 100 101 110 111	C0 C1 C2 C3 C4 C5 C6 C7	C8 C9 CA CB CC CD CE CF	D0 D1 D2 D3 D4 D5 D6 D7	D8 D9 DA DB DC DD DE DF	E0 E1 E2 E3 E4 E5 E6 E7	E8 E9 EA EB EC ED EE EF	F0 F1 F2 F3 F4 F5 F6 F7	F8 F9 FA FB FC FD FE FF

NOTES:

1. The default segment register is SS for the effective addresses containing a BP index, DS for other effective addresses.
2. The disp16 nomenclature denotes a 16-bit displacement that follows the ModR/M byte and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte and that is sign-extended and added to the index.

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX] [ECX] [EDX] [EBX] [--][--][1] disp32[2] [ESI] [EDI]	00	000 001 010 011 100 101 110 111	00 01 02 03 04 05 06 07	08 09 0A 0B 0C 0D 0E 0F	10 11 12 13 14 15 16 17	18 19 1A 1B 1C 1D 1E 1F	20 21 22 23 24 25 26 27	28 29 2A 2B 2C 2D 2E 2F	30 31 32 33 34 35 36 37	38 39 3A 3B 3C 3D 3E 3F
[EAX]+disp8[3] [ECX]+disp8 [EDX]+disp8 [EBX]+disp8 [--][--]+disp8 [EBP]+disp8 [ESI]+disp8 [EDI]+disp8	01	000 001 010 011 100 101 110 111	40 41 42 43 44 45 46 47	48 49 4A 4B 4C 4D 4E 4F	50 51 52 53 54 55 56 57	58 59 5A 5B 5C 5D 5E 5F	60 61 62 63 64 65 66 67	68 69 6A 6B 6C 6D 6E 6F	70 71 72 73 74 75 76 77	78 79 7A 7B 7C 7D 7E 7F
[EAX]+disp32 [ECX]+disp32 [EDX]+disp32 [EBX]+disp32 [--][--]+disp32 [EBP]+disp32 [ESI]+disp32 [EDI]+disp32	10	000 001 010 011 100 101 110 111	80 81 82 83 84 85 86 87	88 89 8A 8B 8C 8D 8E 8F	90 91 92 93 94 95 96 97	98 99 9A 9B 9C 9D 9E 9F	A0 A1 A2 A3 A4 A5 A6 A7	A8 A9 AA AB AC AD AE AF	B0 B1 B2 B3 B4 B5 B6 B7	B8 B9 BA BB BC BD BE BF
EAX/AX/AL/MM0/XMM0 ECX/CX/CL/MM/XMM1 EDX/DX/DL/MM2/XMM2 EBX/BX/BL/MM3/XMM3 ESP/SP/AH/MM4/XMM4 EBP/BP/CH/MM5/XMM5 ESI/SI/DH/MM6/XMM6 EDI/DI/BH/MM7/XMM7	11	000 001 010 011 100 101 110 111	C0 C1 C2 C3 C4 C5 C6 C7	C8 C9 CA CB CC CD CE CF	D0 D1 D2 D3 D4 D5 D6 D7	D8 D9 DA DB DC DD DE DF	E0 E1 E2 E3 E4 E5 E6 E7	E8 E9 EA EB EC ED EE EF	F0 F1 F2 F3 F4 F5 F6 F7	F8 F9 FA FB FC FD FE FF

NOTES:

1. The [--][--] nomenclature means a SIB follows the ModR/M byte.
2. The disp32 nomenclature denotes a 32-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is sign-extended and added to the index.

Table 2-3 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. Table rows in the body of the table indicate the register used as the index (SIB byte bits 3, 4, and 5) and the scaling factor (determined by SIB byte bits 6 and 7).

Table 2-3. 32-Bit Addressing Forms with the SIB Byte

r32 (In decimal) Base = (In binary) Base =			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	**SS**	**Index**	**Value of SIB Byte (in Hexadecimal)**							
[EAX] [ECX] [EDX] [EBX] none [EBP] [ESI] [EDI]	00	000 001 010 011 100 101 110 111	00 08 10 18 20 28 30 38	01 09 11 19 21 29 31 39	02 0A 12 1A 22 2A 32 3A	03 0B 13 1B 23 2B 33 3B	04 0C 14 1C 24 2C 34 3C	05 0D 15 1D 25 2D 35 3D	06 0E 16 1E 26 2E 36 3E	07 0F 17 1F 27 2F 37 3F
[EAX*2] [ECX*2] [EDX*2] [EBX*2] none [EBP*2] [ESI*2] [EDI*2]	01	000 001 010 011 100 101 110 111	40 48 50 58 60 68 70 78	41 49 51 59 61 69 71 79	42 4A 52 5A 62 6A 72 7A	43 4B 53 5B 63 6B 73 7B	44 4C 54 5C 64 6C 74 7C	45 4D 55 5D 65 6D 75 7D	46 4E 56 5E 66 6E 76 7E	47 4F 57 5F 67 6F 77 7F
[EAX*4] [ECX*4] [EDX*4] [EBX*4] none [EBP*4] [ESI*4] [EDI*4]	10	000 001 010 011 100 101 110 111	80 88 90 98 A0 A8 B0 B8	81 89 91 99 A1 A9 B1 B9	82 8A 92 9A A2 AA B2 BA	83 8B 93 9B A3 AB B3 BB	84 8C 94 9C A4 AC B4 BC	85 8D 95 9D A5 AD B5 BD	86 8E 96 9E A6 AE B6 BE	87 8F 97 9F A7 AF B7 BF
[EAX*8] [ECX*8] [EDX*8] [EBX*8] none [EBP*8] [ESI*8] [EDI*8]	11	000 001 010 011 100 101 110 111	C0 C8 D0 D8 E0 E8 F0 F8	C1 C9 D1 D9 E1 E9 F1 F9	C2 CA D2 DA E2 EA F2 FA	C3 CB D3 DB E3 EB F3 FB	C4 CC D4 DC E4 EC F4 FC	C5 CD D5 DD E5 ED F5 FD	C6 CE D6 DE E6 EE F6 FE	C7 CF D7 DF E7 EF F7 FF

NOTES:

1. The [*] nomenclature means a disp32 with no base if the MOD is 00B. Otherwise, [*] means disp8 or disp32 + [EBP]. This provides the following address modes:

MOD bits	Effective Address
00	[scaled index] + disp32
01	[scaled index] + disp8 + [EBP]
10	[scaled index] + disp32 + [EBP]

2.2 IA-32E MODE

IA-32e mode has two sub-modes. These are:

- **Compatibility Mode.** Enables a 64-bit operating system to run most legacy protected mode software unmodified.
- **64-Bit Mode.** Enables a 64-bit operating system to run applications written to access 64-bit address space.

2.2.1 REX Prefixes

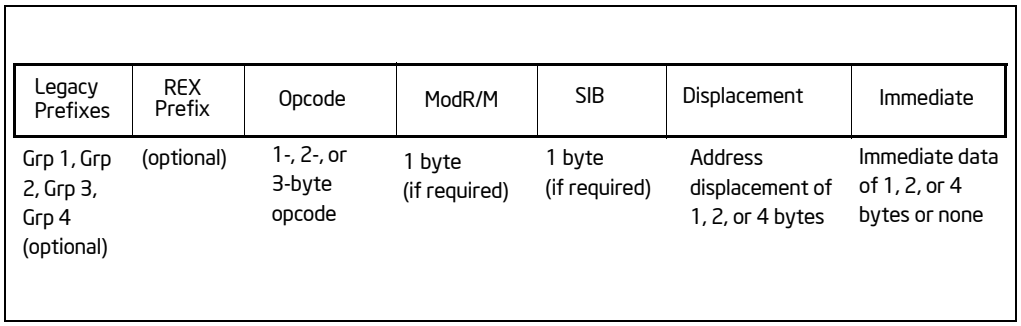
REX prefixes are instruction-prefix bytes used in 64-bit mode. They do the following:

- Specify GPRs and SSE registers.

- Specify 64-bit operand size.
- Specify extended control registers.

Not all instructions require a REX prefix in 64-bit mode. A REX prefix is necessary only if an instruction references one of the extended registers or one of the byte registers SPL, BPL, SIL, DIL; or uses a 64-bit operand. A REX prefix is ignored, as are its individual bits, when it is not needed for an instruction or when it does not immediately precede the opcode byte or the escape opcode byte (0FH) of an instruction for which it is needed. This has the implication that only one REX prefix, properly located, can affect an instruction.

When a REX prefix is used in conjunction with an instruction containing a mandatory prefix, the mandatory prefix must come before the REX so the REX prefix can immediately precede the opcode or the escape byte. For example, CVTDQ2PD with a REX prefix should have REX placed between F3 and 0F E6. Other placements are ignored. The instruction-size limit of 15 bytes still applies to instructions with a REX prefix. See Figure 2-3.



Legacy Prefixes	REX Prefix	Opcode	ModR/M	SIB	Displacement	Immediate
Grp 1, Grp 2, Grp 3, Grp 4 (optional)	(optional)	1-, 2-, or 3-byte opcode	1 byte (if required)	1 byte (if required)	Address displacement of 1, 2, or 4 bytes	Immediate data of 1, 2, or 4 bytes or none

Figure 2-3. Prefix Ordering in 64-bit Mode

2.2.1.1 Encoding

Intel 64 and IA-32 instruction formats specify up to three registers by using 3-bit fields in the encoding, depending on the format:

- ModR/M: the reg and r/m fields of the ModR/M byte.
- ModR/M with SIB: the reg field of the ModR/M byte, the base and index fields of the SIB (scale, index, base) byte.
- Instructions without ModR/M: the reg field of the opcode.

In 64-bit mode, these formats do not change. Bits needed to define fields in the 64-bit context are provided by the addition of REX prefixes.

2.2.1.2 More on REX Prefix Fields

REX prefixes are a set of 16 opcodes that span one row of the opcode map and occupy entries 40H to 4FH. These opcodes represent valid instructions (INC or DEC) in IA-32 operating modes and in compatibility mode. In 64-bit mode, the same opcodes represent the instruction prefix REX and are not treated as individual instructions.

The single-byte-opcode forms of the INC/DEC instructions are not available in 64-bit mode. INC/DEC functionality is still available using ModR/M forms of the same instructions (opcodes FF/0 and FF/1).

See Table 2-4 for a summary of the REX prefix format. Figure 2-4 though Figure 2-7 show examples of REX prefix fields in use. Some combinations of REX prefix fields are invalid. In such cases, the prefix is ignored. Some additional information follows:

- Setting REX.W can be used to determine the operand size but does not solely determine operand width. Like the 66H size prefix, 64-bit operand size override has no effect on byte-specific operations.
- For non-byte operations: if a 66H prefix is used with prefix (REX.W = 1), 66H is ignored.
- If a 66H override is used with REX and REX.W = 0, the operand size is 16 bits.
- REX.R modifies the ModR/M reg field when that field encodes a GPR, SSE, control or debug register. REX.R is ignored when ModR/M specifies other registers or defines an extended opcode.
- REX.X bit modifies the SIB index field.

- REX.B either modifies the base in the ModR/M r/m field or SIB base field; or it modifies the opcode reg field used for accessing GPRs.

Table 2-4. REX Prefix Fields [BITS: 0100WRXB]

Field Name	Bit Position	Definition
-	7:4	0100
W	3	0 = Operand size determined by CS.D
		1 = 64 Bit Operand Size
R	2	Extension of the ModR/M reg field
X	1	Extension of the SIB index field
B	0	Extension of the ModR/M r/m field, SIB base field, or Opcode reg field

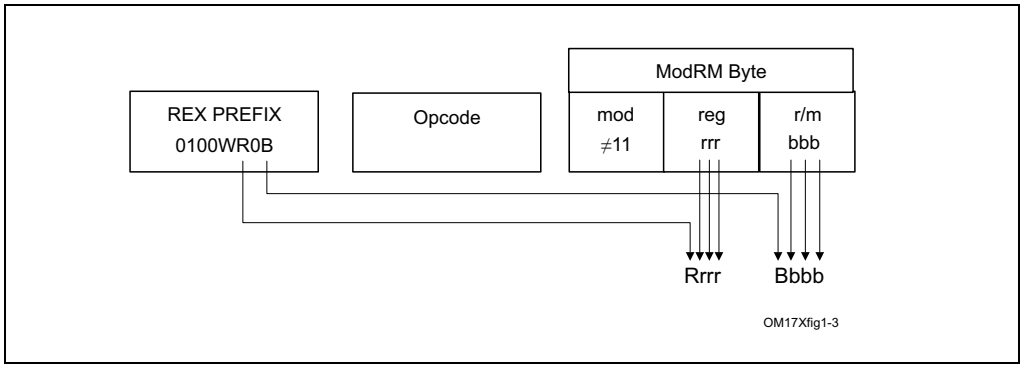


Figure 2-4. Memory Addressing Without an SIB Byte; REX.X Not Used

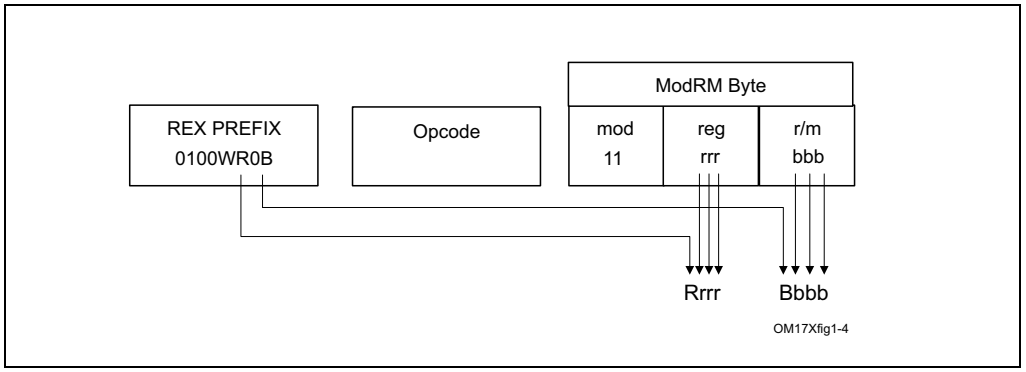


Figure 2-5. Register-Register Addressing (No Memory Operand); REX.X Not Used

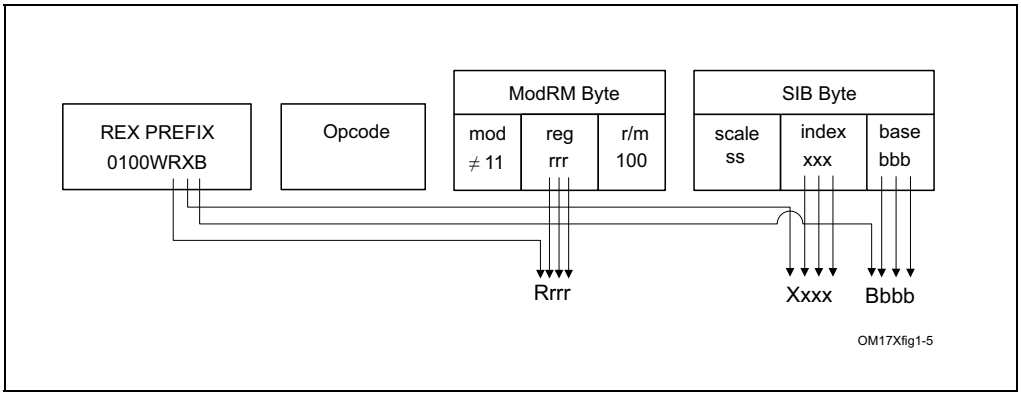


Figure 2-6. Memory Addressing With a SIB Byte

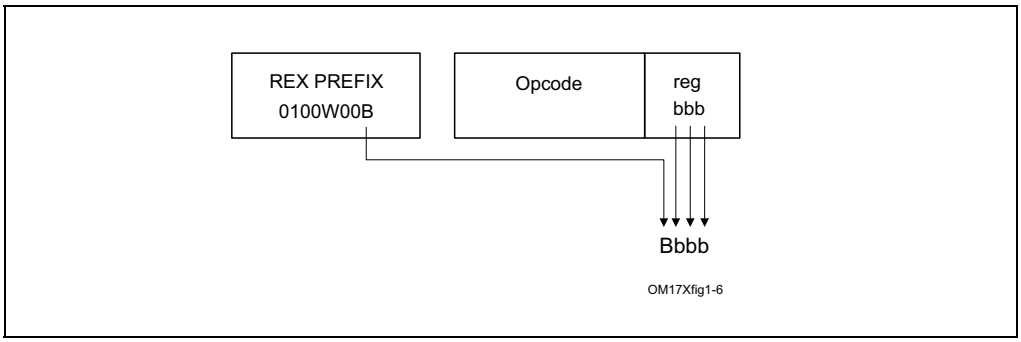


Figure 2-7. Register Operand Coded in Opcode Byte; REX.X & REX.R Not Used

In the IA-32 architecture, byte registers (AH, AL, BH, BL, CH, CL, DH, and DL) are encoded in the ModR/M byte's reg field, the r/m field or the opcode reg field as registers 0 through 7. REX prefixes provide an additional addressing capability for byte-registers that makes the least-significant byte of GPRs available for byte operations.

Certain combinations of the fields of the ModR/M byte and the SIB byte have special meaning for register encodings. For some combinations, fields expanded by the REX prefix are not decoded. Table 2-5 describes how each case behaves.

Table 2-5. Special Cases of REX Encodings

ModR/M or SIB	Sub-field Encodings	Compatibility Mode Operation	Compatibility Mode Implications	Additional Implications
ModR/M Byte	mod ? 11 r/m = b*100(ESP)	SIB byte present.	SIB byte required for ESP-based addressing.	REX prefix adds a fourth bit (b) which is not decoded (don't care). SIB byte also required for R12-based addressing.
ModR/M Byte	mod = 0 r/m = b*101(EBP)	Base register not used.	EBP without a displacement must be done using mod = 01 with displacement of 0.	REX prefix adds a fourth bit (b) which is not decoded (don't care). Using RBP or R13 without displacement must be done using mod = 01 with a displacement of 0.
SIB Byte	index = 0100(ESP)	Index register not used.	ESP cannot be used as an index register.	REX prefix adds a fourth bit (b) which is decoded. There are no additional implications. The expanded index field allows distinguishing RSP from R12, therefore R12 can be used as an index.

Table 2-5. Special Cases of REX Encodings (Contd.)

ModR/M or SIB	Sub-field Encodings	Compatibility Mode Operation	Compatibility Mode Implications	Additional Implications
SIB Byte	base = 0101(EBP)	Base register is unused if mod = 0.	Base register depends on mod encoding.	REX prefix adds a fourth bit (b) which is not decoded. This requires explicit displacement to be used with EBP/RBP or R13.

NOTES:

* Don't care about value of REX.B

2.2.1.3 Displacement

Addressing in 64-bit mode uses existing 32-bit ModR/M and SIB encodings. The ModR/M and SIB displacement sizes do not change. They remain 8 bits or 32 bits and are sign-extended to 64 bits.

2.2.1.4 Direct Memory-Offset MOVs

In 64-bit mode, direct memory-offset forms of the MOV instruction are extended to specify a 64-bit immediate absolute address. This address is called a moffset. No prefix is needed to specify this 64-bit memory offset. For these MOV instructions, the size of the memory offset follows the address-size default (64 bits in 64-bit mode). See Table 2-6.

Table 2-6. Direct Memory Offset Form of MOV

Opcode	Instruction
A0	MOV AL, moffset
A1	MOV EAX, moffset
A2	MOV moffset, AL
A3	MOV moffset, EAX

2.2.1.5 Immediates

In 64-bit mode, the typical size of immediate operands remains 32 bits. When the operand size is 64 bits, the processor sign-extends all immediates to 64 bits prior to their use.

Support for 64-bit immediate operands is accomplished by expanding the semantics of the existing move (MOV reg, imm16/32) instructions. These instructions (opcodes B8H – BFH) move 16-bits or 32-bits of immediate data (depending on the effective operand size) into a GPR. When the effective operand size is 64 bits, these instructions can be used to load an immediate into a GPR. A REX prefix is needed to override the 32-bit default operand size to a 64-bit operand size.

For example:

```
48 B8  8877665544332211  MOV RAX,1122334455667788H
```

2.2.1.6 RIP-Relative Addressing

A new addressing form, RIP-relative (relative instruction-pointer) addressing, is implemented in 64-bit mode. An effective address is formed by adding displacement to the 64-bit RIP of the next instruction.

In IA-32 architecture and compatibility mode, addressing relative to the instruction pointer is available only with control-transfer instructions. In 64-bit mode, instructions that use ModR/M addressing can use RIP-relative addressing. Without RIP-relative addressing, all ModR/M modes address memory relative to zero.

RIP-relative addressing allows specific ModR/M modes to address memory relative to the 64-bit RIP using a signed 32-bit displacement. This provides an offset range of ±2GB from the RIP. Table 2-7 shows the ModR/M and SIB encodings for RIP-relative addressing. Redundant forms of 32-bit displacement-addressing exist in the current ModR/M and SIB encodings. There is one ModR/M encoding and there are several SIB encodings. RIP-relative addressing is encoded using a redundant form.

In 64-bit mode, the ModR/M Disp32 (32-bit displacement) encoding is re-defined to be RIP+Disp32 rather than displacement-only. See Table 2-7.

Table 2-7. RIP-Relative Addressing

ModR/M and SIB Sub-field Encodings		Compatibility Mode Operation	64-bit Mode Operation	Additional Implications in 64-bit mode
ModR/M Byte	mod = 00 r/m = 101 (none)	Disp32	RIP + Disp32	In 64-bit mode, if one wants to use a Disp32 without specifying a base register, one can use a SIB byte encoding (indicated by ModR/M.r/m=100) as described in the next row.
SIB Byte	base = 101 (none) index = 100 (none) scale = 0, 1, 2, 4	If mod = 00, Disp32	Same as legacy	None

The ModR/M encoding for RIP-relative addressing does not depend on using a prefix. Specifically, the r/m bit field encoding of 101B (used to select RIP-relative addressing) is not affected by the REX prefix. For example, selecting R13 (REX.B = 1, r/m = 101B) with mod = 00B still results in RIP-relative addressing. The 4-bit r/m field of REX.B combined with ModR/M is not fully decoded. In order to address R13 with no displacement, software must encode R13 + 0 using a 1-byte displacement of zero.

RIP-relative addressing is enabled by 64-bit mode, not by a 64-bit address-size. The use of the address-size prefix does not disable RIP-relative addressing. The effect of the address-size prefix is to truncate and zero-extend the computed effective address to 32 bits.

2.2.1.7 Default 64-Bit Operand Size

In 64-bit mode, two groups of instructions have a default operand size of 64 bits (do not need a REX prefix for this operand size). These are:

- Near branches.
- All instructions, except far branches, that implicitly reference the RSP.

2.2.2 Additional Encodings for Control and Debug Registers

In 64-bit mode, more encodings for control and debug registers are available. The REX.R bit is used to modify the ModR/M reg field when that field encodes a control or debug register (see Table 2-4). These encodings enable the processor to address CR8-CR15 and DR8- DR15. An additional control register (CR8) is defined in 64-bit mode. CR8 becomes the Task Priority Register (TPR).

In the first implementation of IA-32e mode, CR9-CR15 and DR8-DR15 are not implemented. Any attempt to access unimplemented registers results in an invalid-opcode exception (#UD).

2.3 INTEL® ADVANCED VECTOR EXTENSIONS (INTEL® AVX)

Intel AVX instructions are encoded using an encoding scheme that combines prefix bytes, opcode extension field, operand encoding fields, and vector length encoding capability into a new prefix, referred to as VEX. In the VEX encoding scheme, the VEX prefix may be two or three bytes long, depending on the instruction semantics. Despite the two-byte or three-byte length of the VEX prefix, the VEX encoding format provides a more compact representation/packing of the components of encoding an instruction in Intel 64 architecture. The VEX encoding scheme also allows more headroom for future growth of Intel 64 architecture.

2.3.1 Instruction Format

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 2-8 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.

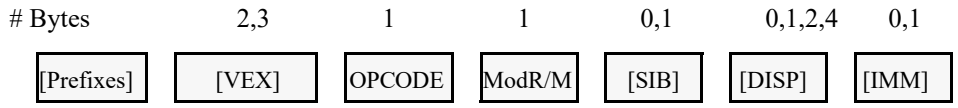


Figure 2-8. Instruction Encoding Format with VEX Prefix

2.3.2 VEX and the LOCK prefix

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

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

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

2.3.4 VEX and the REX prefix

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

2.3.5 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 2-9.

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.

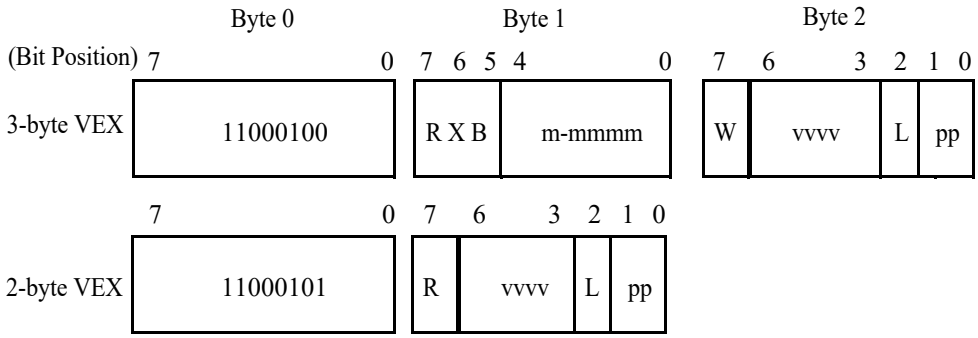
The VEX prefix is required to be the last prefix and immediately precedes the opcode bytes. It must follow any other prefixes. If VEX prefix is present a REX prefix is not supported.

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. Note, certain new instruction functionality can only be encoded with the VEX prefix.

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



R: REX.R in 1's complement (inverted) form
1: Same as REX.R=0 (must be 1 in 32-bit mode)
0: Same as REX.R=1 (64-bit mode only)

X: REX.X in 1's complement (inverted) form
1: Same as REX.X=0 (must be 1 in 32-bit mode)
0: Same as REX.X=1 (64-bit mode only)

B: REX.B in 1's complement (inverted) form
1: Same as REX.B=0 (Ignored in 32-bit mode).
0: Same as REX.B=1 (64-bit mode only)

W: opcode specific (use like REX.W, or used for 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 2-9. VEX bit fields

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

2.3.5.1 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.

2.3.5.2 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.

2.3.5.3 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.

2.3.5.4 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.

2.3.5.5 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 for these instructions, 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.

2.3.5.6 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 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 2-8. VEX.vvvv to Register Name Mapping

VEX.vvvv	Dest Register	General-Purpose Register (If Applicable)[1]	Valid in Legacy/Compatibility 32-bit modes?[2]
1111B	XMM0/YMM0	RAX/EAX	Valid
1110B	XMM1/YMM1	RCX/ECX	Valid
1101B	XMM2/YMM2	RDX/EDX	Valid
1100B	XMM3/YMM3	RBX/EBX	Valid
1011B	XMM4/YMM4	RSP/ESP	Valid
1010B	XMM5/YMM5	RBP/EBP	Valid
1001B	XMM6/YMM6	RSI/ESI	Valid
1000B	XMM7/YMM7	RDI/EDI	Valid
0111B	XMM8/YMM8	R8/R8D	Invalid
0110B	XMM9/YMM9	R9/R9D	Invalid
0101B	XMM10/YMM10	R10/R10D	Invalid
0100B	XMM11/YMM11	R11/R11D	Invalid
0011B	XMM12/YMM12	R12/R12D	Invalid
0010B	XMM13/YMM13	R13/R13D	Invalid
0001B	XMM14/YMM14	R14/R14D	Invalid
0000B	XMM15/YMM15	R15/R15D	Invalid

NOTES:

1. See Section 2.6, "VEX Encoding Support for GPR Instructions" for additional details.
2. Only the first eight General-Purpose Registers are accessible/encodable in 16/32b modes.

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

2.3.6 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 2-9. The notation in the "Opcode" column in Table 2-9 is described in detail in section 3.1.1.
- VEX.vvvv does not encode any operand, the field is reserved and should contain 1111b.

Table 2-9. Instructions with a VEX.vvvv Destination

Opcode	Instruction mnemonic
VEX.128.66.0F 73 /7 ib	VPSLLDQ xmm1, xmm2, imm8
VEX.128.66.0F 73 /3 ib	VPSRLDQ xmm1, xmm2, imm8

Table 2-9. Instructions with a VEX.vvvv Destination (Contd.)

Opcode	Instruction mnemonic
VEX.128.66.0F 71 /2 ib	VPSRLW xmm1, xmm2, imm8
VEX.128.66.0F 72 /2 ib	VPSRLD xmm1, xmm2, imm8
VEX.128.66.0F 73 /2 ib	VPSRLQ xmm1, xmm2, imm8
VEX.128.66.0F 71 /4 ib	VPSRAW xmm1, xmm2, imm8
VEX.128.66.0F 72 /4 ib	VPSRAD xmm1, xmm2, imm8
VEX.128.66.0F 71 /6 ib	VPSLLW xmm1, xmm2, imm8
VEX.128.66.0F 72 /6 ib	VPSLLD xmm1, xmm2, imm8
VEX.128.66.0F 73 /6 ib	VPSLLQ xmm1, xmm2, imm8

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

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

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

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

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

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

2.3.6.1 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 2-10. 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.

2.3.6.2 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 Table 2-11.

Table 2-11. VEX.L Interpretation

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

2.3.6.3 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 Table 2-12.

Table 2-12. VEX.pp Interpretation

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

2.3.7 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.

2.3.8 The ModR/M, SIB, and Displacement Bytes

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

2.3.9 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.

2.3.10 Intel® 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 bits (above bit 128) of the equivalent YMM register. Legacy SSE instructions without VEX preserve the upper bits.

2.3.10.1 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.)

2.3.11 Intel® AVX Instruction Length

The Intel AVX 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.

2.3.12 Vector SIB (VSIB) Memory Addressing

In Intel® Advanced Vector Extensions 2 (Intel® 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 Intel 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 2-13 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 R8D-R15D applicable only in 64-bit mode (when address size override prefix is used, but the value of VEX.B is not shown in Table 2-13). In 32-bit mode, R8D-R15D 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 2-13. 32-Bit VSIB Addressing Forms of the SIB Byte

r32 (In decimal) Base = (In binary) Base =				EAX/ R8D 0 000	ECX/ R9D 1 001	EDX/ R10D 2 010	EBX/ R11D 3 011	ESP/ R12D 4 100	EBP/ R13D[1] 5 101	ESI/ R14D 6 110	EDI/ R15D 7 111
Scaled Index		**SS**	**Index**	**Value of SIB Byte (in Hexadecimal)**							
VR0/VR8 VR1/VR9 VR2/VR10 VR3/VR11 VR4/VR12 VR5/VR13 VR6/VR14 VR7/VR15	*1	00	000 001 010 011 100 101 110 111	00 08 10 18 20 28 30 38	01 09 11 19 21 29 31 39	02 0A 12 1A 22 2A 32 3A	03 0B 13 1B 23 2B 33 3B	04 0C 14 1C 24 2C 34 3C	05 0D 15 1D 25 2D 35 3D	06 0E 16 1E 26 2E 36 3E	07 0F 17 1F 27 2F 37 3F
VR0/VR8 VR1/VR9 VR2/VR10 VR3/VR11 VR4/VR12 VR5/VR13 VR6/VR14 VR7/VR15	*2	01	000 001 010 011 100 101 110 111	40 48 50 58 60 68 70 78	41 49 51 59 61 69 71 79	42 4A 52 5A 62 6A 72 7A	43 4B 53 5B 63 6B 73 7B	44 4C 54 5C 64 6C 74 7C	45 4D 55 5D 65 6D 75 7D	46 4E 56 5E 66 6E 76 7E	47 4F 57 5F 67 6F 77 7F
VR0/VR8 VR1/VR9 VR2/VR10 VR3/VR11 VR4/VR12 VR5/VR13 VR6/VR14 VR7/VR15	*4	10	000 001 010 011 100 101 110 111	80 88 90 98 A0 A8 B0 B8	81 89 91 89 A1 A9 B1 B9	82 8A 92 9A A2 AA B2 BA	83 8B 93 9B A3 AB B3 BB	84 8C 94 9C A4 AC B4 BC	85 8D 95 9D A5 AD B5 BD	86 8E 96 9E A6 AE B6 BE	87 8F 97 9F A7 AF B7 BF
VR0/VR8 VR1/VR9 VR2/VR10 VR3/VR11 VR4/VR12 VR5/VR13 VR6/VR14 VR7/VR15	*8	11	000 001 010 011 100 101 110 111	C0 C8 D0 D8 E0 E8 F0 F8	C1 C9 D1 D9 E1 E9 F1 F9	C2 CA D2 DA E2 EA F2 FA	C3 CB D3 DB E3 EB F3 FB	C4 CC D4 DC E4 EC F4 FC	C5 CD D5 DD E5 ED F5 FD	C6 CE D6 DE E6 EE F6 FE	C7 CF D7 DF E7 EF F7 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]

2.3.12.1 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 2-13 base register should be interpreted as the full 64-bit of each register.

2.4 INTEL® ADVANCED MATRIX EXTENSIONS (INTEL® AMX)

Intel® AMX instructions follow the general documentation convention established in previous sections. Additionally, Intel® Advanced Matrix Extensions use notation conventions as described below.

In the instruction encoding boxes, **sibmem** is used to denote an encoding where a ModR/M byte and SIB byte are used to indicate a memory operation where the base and displacement are used to point to memory, and the index

register (if present) is used to denote a stride between memory rows. The index register is scaled by the sib.scale field as usual. The base register is added to the displacement, if present.

In the instruction encoding, the ModR/M byte is represented several ways depending on the role it plays. The ModR/M byte has 3 fields: 2-bit ModR/M.mod field, a 3-bit ModR/M.reg field and a 3-bit ModR/M.r/m field. When all bits of the ModR/M byte have fixed values for an instruction, the 2-hex nibble value of that byte is presented after the opcode in the encoding boxes on the instruction description pages. When only some fields of the ModR/M byte must contain fixed values, those values are specified as follows:

- If only the ModR/M.mod must be 0b11, and ModR/M.reg and ModR/M.r/m fields are unrestricted, this is denoted as **11:rrr:bbb**. The **rrr** correspond to the 3-bits of the ModR/M.reg field and the **bbb** correspond to the 3-bits of the ModR/M.r/m field.
- If the ModR/M.mod field is constrained to be a value other than 0b11, i.e., it must be one of 0b00, 0b01, or 0b10, then the notation !(11) is used.
- If the ModR/M.reg field had a specific required value, e.g., 0b101, that would be denoted as mm:101:bbb.

NOTE

Historically this document only specified the ModR/M.reg field restrictions with the notation /0 ... /7 and did not specify restrictions on the ModR/M.mod and ModR/M.r/m fields in the encoding boxes.

2.5 INTEL® AVX AND INTEL® SSE INSTRUCTION EXCEPTION CLASSIFICATION

To look up the exceptions of legacy 128-bit SIMD instruction, 128-bit VEX-encoded instructions, and 256-bit VEX-encoded instruction, Table summarizes the exception behavior into separate classes, with detailed exception conditions defined in sub-sections 2.5.1 through 2.6.1. For example, ADDPS contains the entry:

"See Exceptions Type 2."

In this entry, "Type 2" can be looked up in Table 2-19.

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.

NOTE

Instructions that operate only with MMX, X87, or general-purpose registers are not covered by the exception classes defined in this section. For instructions that operate on MMX registers, see Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Table 2-14. Exception Class Description

Exception Class	Instruction Set	Mem Arg	Floating-Point Exceptions (#XM)
Type 1	AVX, Legacy SSE	16/32 byte explicitly aligned	No
Type 2	AVX, Legacy SSE	16/32 byte not explicitly aligned	Yes
Type 3	AVX, Legacy SSE	< 16 byte	Yes
Type 4	AVX, Legacy SSE	16/32 byte not explicitly aligned	No
Type 5	AVX, Legacy SSE	< 16 byte	No
Type 6	AVX (no Legacy SSE)	Varies	(At present, none do)

Table 2-14. Exception Class Description (Contd.)

Exception Class	Instruction Set	Mem Arg	Floating-Point Exceptions (#XM)
Type 7	AVX, Legacy SSE	None	No
Type 8	AVX	None	No
Type 11	F16C	8 or 16 byte, Not explicitly aligned, no AC#	Yes
Type 12	AVX2 Gathers	Not explicitly aligned, no AC#	No

See Table 2-15 for lists of instructions in each exception class.

Table 2-15. 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)ROUNDPD, (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, (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)PSUBUSB, (V)PSUBUSW, (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, VPERMILPD, VPERMILPS, VPERM2F128

Table 2-15. Instructions in Each Exception Class (Contd.)

Exception Class	Instruction
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, PMOVSXBW, (V)RCPSS, (V)RSQRTSS, (V)PMOVSX/ZX, VLDMXCSR*, VSTMXCSR
Type 6	VEXTRACTF128/VEXTRACTFxxxx, 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.

(***) - PCMPESTRI, PCMPESTRM, PCMPISTRI, PCMPISTRM, and LDDQU instructions do not cause #GP if the memory operand is not aligned to 16-Byte boundary.

Table 2-15 classifies exception behaviors for Intel AVX instructions. Within each class of exception conditions that are listed in Table 2-18 through Table 2-27, certain subsets of Intel AVX instructions may be subject to #UD exception depending on the encoded value of the VEX.L field. Table 2-16 and Table 2-17 provide supplemental information of Intel AVX instructions that may be subject to #UD exception if encoded with incorrect values in the VEX.W or VEX.L field.

Table 2-16. #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, VPERMILPD, VPERMILPS, VPERM2F128	
Type 5		
Type 6	VEXTRACTF128, VBROADCASTSS, VBROADCASTSD, VBROADCASTF128, VINSERTF128, VMASKMOVPS, VMASKMOVPD, VBROADCASTI128, VPBROADCASTB/W/D, VEXTRACTI128, VINSERTI128	
Type 7		
Type 8		
Type 11	VCVTPH2PS, VCVTPS2PH	
Type 12		

Table 2-17. #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			
Type 4		VMASKMOVDQU, VMPSADBW, VPABSB/W/D, VPACKSSWB/DW, VPACKUSWB/DW, VPADDB/W/D, VPADDQ, VPADDSB/W, VPADDUSB/W, VPALIGNR, VPAND, VPANDN, VPAVGB/W, VPBLENDVB, VPBLENDW, VPCMP(E/I)STRI/M, VPCMPEQB/W/D/Q, VPCMPGTB/W/D/Q, VPHADDW/D, VPHADDSW, VPHMINPOSUW, VPHSUBD/W, VPHSUBSW, VPMADDWD, VPMADDUBSW, 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, VPUNPCKHBW/WD/DQ, VPUNPCKHQDQ, 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			
Type 11			
Type 12			

2.5.1 Exceptions Type 1 (Aligned Memory Reference)

Table 2-18. Type 1 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCR0[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.

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

Table 2-19. 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 XCR0[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.

2.5.3 Exceptions Type 3 (<16 Byte Memory Argument)

Table 2-20. Type 3 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 XCR0[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	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access 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.

2.5.4 Exceptions Type 4 (>=16 Byte Mem Arg, No Alignment, No Floating-point Exceptions)

Table 2-21. Type 4 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCR0[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.[1]
			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.

NOTES:

1. LDDQU, MOVUPD, MOVUPS, PCMPESTRI, PCMPESTRM, PCMPISTRI, and PCMPISTRM instructions do not cause #GP if the memory operand is not aligned to 16-Byte boundary.

2.5.5 Exceptions Type 5 (<16 Byte Mem Arg and No FP Exceptions)

Table 2-22. Type 5 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCR0[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	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

2.5.6 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 2-23. Type 6 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	If XCR0[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 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

2.5.7 Exceptions Type 7 (No FP Exceptions, No Memory Arg)

Table 2-24. Type 7 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCR0[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.

2.5.8 Exceptions Type 8 (AVX and No Memory Argument)

Table 2-25. Type 8 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			Always in Real or Virtual-8086 mode.
			X	X	If XCR0[2:1] ? '11b'. If CR4.OSXSAVE[bit 18]=0. If CPUID.01H:ECX.AVX[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.

2.5.9 Exceptions Type 11 (VEX-only, Mem Arg, No AC, Floating-point Exceptions)

Table 2-26. Type 11 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCR0[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.

2.5.10 Exceptions Type 12 (VEX-only, VSIB Mem Arg, No AC, No Floating-point Exceptions)

Table 2-27. Type 12 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCR0[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.

2.6 VEX ENCODING SUPPORT FOR GPR INSTRUCTIONS

The 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.

2.6.1 Exceptions Type 13 (VEX-Encoded GPR Instructions)

The exception conditions applicable to VEX-encoded GPR instructions differ from those of legacy GPR instructions. Table 2-28 lists VEX-encoded GPR instructions. The exception conditions for VEX-encoded GPR instructions are found in Table 2-29 for those instructions which have a default operand size of 32 bits and 16-bit operand size is not encodable.

Table 2-28. VEX-Encoded GPR Instructions

Exception Class	Instruction
Type 13	ANDN, BEXTR, BLSI, BLSMSK, BLSR, BZHI, MULX, PDEP, PEXT, RORX, SARX, SHLX, SHRX

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

Table 2-29. Type 13 Class Exception Conditions

Exception	Real	Virtual-8086	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	X	X	If VEX.L = 1.
	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.
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	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

2.6.2 Exceptions Type 14 (CMPCCXADD)

The exception conditions applicable to the CMPCCXADD instruction differ from those of other VEX-encoded GPR instructions. The exception conditions for the CMPCCXADD instruction are found in Table 2-31.

Table 2-30. Exceptions Type 14 Instructions

Exception Class	Instruction
Type 14	CMPCCXADD

Table 2-31. Type 14 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X		Only supported in 64-bit mode.
				X	If any LOCK, REX, F2, F3, or 66 prefixes precede a VEX prefix.
				X	If any corresponding CPUID feature flag is '0'.
Stack, #SS(0)				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)				X	If the memory address is in a non-canonical form.
Page Fault, #PF(fault-code)				X	If a page fault occurs.
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 INTEL® AVX-512 ENCODING

The majority of the Intel AVX-512 family of instructions (operating on 512/256/128-bit vector register operands) are encoded using a new prefix (called EVEX). Opmask instructions (operating on opmask register operands) are encoded using the VEX prefix. The EVEX prefix has some parts resembling the instruction encoding scheme using the VEX prefix, and many other capabilities not available with the VEX prefix.

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

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

2.7.1 Instruction Format and EVEX

The placement of the EVEX prefix in an IA instruction is represented in Figure 2-10. Note that the values contained within brackets are optional.

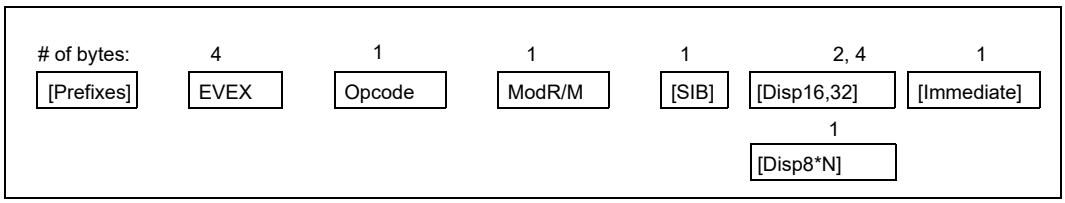


Figure 2-10. Intel® AVX-512 Instruction Format and the EVEX Prefix

The EVEX prefix is a 4-byte prefix, with the first two bytes derived from unused encoding form of the 32-bit-mode-only BOUND instruction. The layout of the EVEX prefix is shown in Figure 2-11. The first byte must be 62H, followed by three payload bytes, denoted as P0, P1, and P2 individually or collectively as P[23:0] (see Figure 2-11).

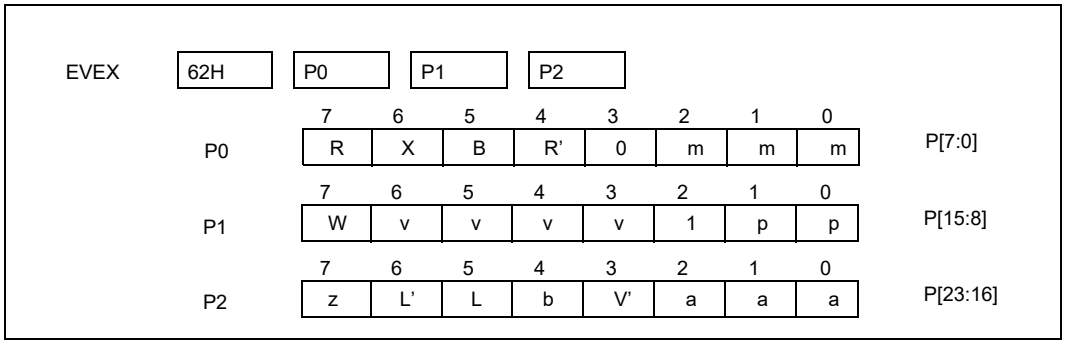


Figure 2-11. Bit Field Layout of the EVEX Prefix[1]

NOTES:

1. See Table 2-32 for additional details on bit fields.

Table 2-32. EVEX Prefix Bit Field Functional Grouping

Notation	Bit field Group	Position	Comment
EVEX.mmm	Access to up to eight decoding maps	P[2:0]	Currently, only the following decoding maps are supported: 1, 2, 3, 5, and 6.
--	Reserved	P[3]	Must be 0.
EVEX.R'	High-16 register specifier modifier	P[4]	Combine with EVEX.R and ModR/M.reg. This bit is stored in inverted format.
EVEX.RXB	Next-8 register specifier modifier	P[7:5]	Combine with ModR/M.reg, ModR/M.rm (base, index/vidx). This field is encoded in bit inverted format.
EVEX.X	High-16 register specifier modifier	P[6]	Combine with EVEX.B and ModR/M.rm, when SIB/VSIB absent.
EVEX.pp	Compressed legacy prefix	P[9:8]	Identical to VEX.pp.
--	Fixed Value	P[10]	Must be 1.
EVEX.vvvv	VVVV register specifier	P[14:11]	Same as VEX.vvvv. This field is encoded in bit inverted format.
EVEX.W	Operand size promotion/Opcode extension	P[15]	
EVEX.aaa	Embedded opmask register specifier	P[18:16]	
EVEX.V'	High-16 VVVV/VIDX register specifier	P[19]	Combine with EVEX.vvvv or when VSIB present. This bit is stored in inverted format.
EVEX.b	Broadcast/RC/SAE Context	P[20]	
EVEX.L'L	Vector length/RC	P[22:21]	
EVEX.z	Zeroing/Merging	P[23]	

The bit fields in P[23:0] are divided into the following functional groups (Table 2-32 provides a tabular summary):

- Reserved bits: P[3] must be 0, otherwise #UD.
- Fixed-value bit: P[10] must be 1, otherwise #UD.
- Compressed legacy prefix/escape bytes: P[1:0] is identical to the lowest 2 bits of VEX.mmmmm; P[9:8] is identical to VEX.pp.
- EVEX.mmm: P[2:0] provides access to up to eight decoding maps. Currently, only the following decoding maps are supported: 1, 2, 3, 5, and 6. Map ids 1, 2, and 3 are denoted by 0F, 0F38, and 0F3A, respectively, in the instruction encoding descriptions.
- Operand specifier modifier bits for vector register, general purpose register, memory addressing: P[7:5] allows access to the next set of 8 registers beyond the low 8 registers when combined with ModR/M register specifiers.
- Operand specifier modifier bit for vector register: P[4] (or EVEX.R') allows access to the high 16 vector register set when combined with P[7] and ModR/M.reg specifier; P[6] can also provide access to a high 16 vector register when SIB or VSIB addressing are not needed.
- Non-destructive source /vector index operand specifier: P[19] and P[14:11] encode the second source vector register operand in a non-destructive source syntax, vector index register operand can access an upper 16 vector register using P[19].
- Op-mask register specifiers: P[18:16] encodes op-mask register set k0-k7 in instructions operating on vector registers.
- EVEX.W: P[15] is similar to VEX.W which serves either as opcode extension bit or operand size promotion to 64-bit in 64-bit mode.
- Vector destination merging/zeroing: P[23] encodes the destination result behavior which either zeroes the masked elements or leave masked element unchanged.
- Broadcast/Static-rounding/SAE context bit: P[20] encodes multiple functionality, which differs across different classes of instructions and can affect the meaning of the remaining field (EVEX.L'L). The functionality for the following instruction classes are:

  - Broadcasting a single element across the destination vector register: this applies to the instruction class with Load+Op semantic where one of the source operand is from memory.
  - Redirect L'L field (P[22:21]) as static rounding control for floating-point instructions with rounding semantic. Static rounding control overrides MXCSR.RC field and implies "Suppress all exceptions" (SAE).
  - Enable SAE for floating -point instructions with arithmetic semantic that is not rounding.
  - For instruction classes outside of the afore-mentioned three classes, setting EVEX.b will cause #UD.
- Vector length/rounding control specifier: P[22:21] can serve one of three options.
  - Vector length information for packed vector instructions.
  - Ignored for instructions operating on vector register content as a single data element.
  - Rounding control for floating-point instructions that have a rounding semantic and whose source and destination operands are all vector registers.

2.7.2 Register Specifier Encoding and EVEX

EVEX-encoded instruction can access 8 opmask registers, 16 general-purpose registers and 32 vector registers in 64-bit mode (8 general-purpose registers and 8 vector registers in non-64-bit modes). EVEX-encoding can support instruction syntax that access up to 4 instruction operands. Normal memory addressing modes and VSIB memory addressing are supported with EVEX prefix encoding. The mapping of register operands used by various instruction syntax and memory addressing in 64-bit mode are shown in Table 2-33. Opmask register encoding is described in Section 2.7.3.

Table 2-33. 32-Register Support in 64-bit Mode Using EVEX with Embedded REX Bits

	4[1]	3	[2:0]	Reg. Type	Common Usages
REG	EVEX.R'	REX.R	modrm.reg	GPR, Vector	Destination or Source
VVVV	EVEX.V'	EVEX.vvvv		GPR, Vector	2ndSource or Destination
RM	EVEX.X	EVEX.B	modrm.r/m	GPR, Vector	1st Source or Destination
BASE	0	EVEX.B	modrm.r/m	GPR	memory addressing
INDEX	0	EVEX.X	sib.index	GPR	memory addressing
VIDX	EVEX.V'	EVEX.X	sib.index	Vector	VSIB memory addressing

NOTES:

1. Not applicable for accessing general purpose registers.

The mapping of register operands used by various instruction syntax and memory addressing in 32-bit modes are shown in Table 2-34.

Table 2-34. EVEX Encoding Register Specifiers in 32-bit Mode

	[2:0]	Reg. Type	Common Usages
REG	modrm.reg	GPR, Vector	Destination or Source
VVVV	EVEX.vvv	GPR, Vector	2nd Source or Destination
RM	modrm.r/m	GPR, Vector	1st Source or Destination
BASE	modrm.r/m	GPR	Memory Addressing
INDEX	sib.index	GPR	Memory Addressing
VIDX	sib.index	Vector	VSIB Memory Addressing

2.7.3 Opmask Register Encoding

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

- Opmask registers that are the source or destination operands of an instruction treating the content of opmask register as a scalar value, are encoded using the VEX prefix scheme. It can support up to three operands using standard modR/M byte's reg field and rm field and VEX.vvvv. Such a scalar opmask instruction does not support conditional update of the destination operand.
- An opmask register providing conditional processing and/or conditional update of the destination register of a vector instruction is encoded using EVEX.aaa field (see Section 2.7.4).
- An opmask register serving as the destination or source operand of a vector instruction is encoded using standard modR/M byte's reg field and rm fields.

Table 2-35. Opmask Register Specifier Encoding

	[2:0]	Register Access	Common Usages
REG	modrm.reg	k0-k7	Source
VVVV	VEX.vvvv	k0-k7	2nd Source
RM	modrm.r/m	k0-7	1st Source
{k1}	EVEX.aaa	k0[1]-k7	Opmask

NOTES:

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

2.7.4 Masking Support in EVEX

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

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

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

AVX-512 Foundation instructions can be divided into the following groups:

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

2.7.5 Compressed Displacement (disp8*N) Support in EVEX

For memory addressing using disp8 form, EVEX-encoded instructions always use a compressed displacement scheme by multiplying disp8 in conjunction with a scaling factor N that is determined based on the vector length, the value of EVEX.b bit (embedded broadcast) and the input element size of the instruction. In general, the factor N corresponds to the number of bytes characterizing the internal memory operation of the input operand (e.g., 64 when the accessing a full 512-bit memory vector). The scale factor N is listed in Table 2-36 and Table 2-37 below, where EVEX encoded instructions are classified using the **tupletype** attribute. The scale factor N of each tupletype is listed based on the vector length (VL) and other factors affecting it.

Table 2-36 covers EVEX-encoded instructions which has a load semantic in conjunction with additional computational or data element movement operation, operating either on the full vector or half vector (due to conversion of numerical precision from a wider format to narrower format). EVEX.b is supported for such instructions for data element sizes which are either dword or qword (see Section 2.7.11).

EVEX-encoded instruction that are pure load/store, and "Load+op" instruction semantic that operate on data element size less then dword do not support broadcasting using EVEX.b. These are listed in Table 2-37. Table 2-37 also includes many broadcast instructions which perform broadcast using a subset of data elements without using EVEX.b. These instructions and a few data element size conversion instruction are covered in Table 2-37. Instruction classified in Table 2-37 do not use EVEX.b and EVEX.b must be 0, otherwise #UD will occur.

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

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

Table 2-36. Compressed Displacement (DISP8*N) Affected by Embedded Broadcast

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

Table 2-37. EVEX DISP8*N for Instructions Not Affected by Embedded Broadcast

TupleType	InputSize	EVEX.W	N (VL= 128)	N (VL= 256)	N (VL= 512)	Comment
Full Mem	N/A	N/A	16	32	64	Load/store or subDword full vector
Tuple1 Scalar	8bit	N/A	1	1	1	1Tuple
	16bit	N/A	2	2	2	
	32bit	0	4	4	4	
	64bit	1	8	8	8	
Tuple1 Fixed	32bit	N/A	4	4	4	1 Tuple, memsize not affected by EVEX.W
	64bit	N/A	8	8	8	
Tuple2	32bit	0	8	8	8	Broadcast (2 elements)
	64bit	1	NA	16	16	
Tuple4	32bit	0	NA	16	16	Broadcast (4 elements)
	64bit	1	NA	NA	32	
Tuple8	32bit	0	NA	NA	32	Broadcast (8 elements)
Half Mem	N/A	N/A	8	16	32	SubQword Conversion
Quarter Mem	N/A	N/A	4	8	16	SubDword Conversion

Table 2-37. EVEX DISP8*N for Instructions Not Affected by Embedded Broadcast (Contd.)

TupleType	InputSize	EVEX.W	N (VL= 128)	N (VL= 256)	N (VL= 512)	Comment
Eighth Mem	N/A	N/A	2	4	8	SubWord Conversion
Mem128	N/A	N/A	16	16	16	Shift count from memory
MOVDDUP	N/A	N/A	8	32	64	VMOVDDUP

2.7.6 EVEX Encoding of Broadcast/Rounding/SAE Support

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

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

2.7.7 Embedded Broadcast Support in EVEX

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

The following instruction classes do not support embedded broadcasting:

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

2.7.8 Static Rounding Support in EVEX

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

2.7.9 SAE Support in EVEX

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

2.7.10 Vector Length Orthogonality

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

When EVEX.b bit is set for a register-register instructions with floating-point rounding semantic, the same two bits P2[6:5] specifies rounding mode for the instruction, with implied SAE behavior. The mapping of different instruction classes relative to the embedded broadcast/rounding/SAE control and the EVEX.L'L fields are summarized in Table 2-38.

Table 2-38. EVEX Embedded Broadcast/Rounding/SAE and Vector Length on Vector Instructions

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

2.7.11 #UD Equations for EVEX

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

2.7.11.1 State Dependent #UD

In general, attempts of execute an instruction, which required OS support for incremental extended state component, will #UD if required state components were not enabled by OS. Table 2-39 lists instruction categories with respect to required processor state components. Attempts to execute a given category of instructions while enabled states were less than the required bit vector in XCR0 shown in Table 2-39 will cause #UD.

Table 2-39. OS XSAVE Enabling Requirements of Instruction Categories

Instruction Categories	Vector Register State Access	Required XCR0 Bit Vector [7:0]
Legacy SIMD prefix encoded Instructions (e.g SSE)	XMM	xxxxxx11b
VEX-encoded instructions operating on YMM	YMM	xxxxx111b
EVEX-encoded 128-bit instructions	ZMM	111xx111b
EVEX-encoded 256-bit instructions	ZMM	111xx111b
EVEX-encoded 512-bit instructions	ZMM	111xx111b
VEX-encoded instructions operating on opmask	k-reg	111xxx11b

2.7.11.2 Opcode Independent #UD

A number of bit fields in EVEX encoded instruction must obey mode-specific but opcode-independent patterns listed in Table 2-40.

Table 2-40. Opcode Independent, State Dependent EVEX Bit Fields[1]

Position	Notation	64-bit #UD	Non-64-bit #UD
P[3]	--	if > 0	if > 0
P[10]	--	if 0	if 0
P[2:0]	EVEX.mmm	if 000b, 100b, or 111b	if 000b, 100b, or 111b
P[7 : 6]	EVEX.RX	None (valid)	None (BOUND if EVEX.RX != 11b)

NOTES:

1. This table is also representative of VEX restrictions. For VEX operations, use the Notation field.

2.7.11.3 Opcode Dependent #UD

This section describes legal values for the rest of the EVEX bit fields. Table 2-41 lists the #UD conditions of EVEX prefix bit fields which encodes or modifies register operands.

Table 2-41. #UD Conditions of Operand-Encoding EVEX Prefix Bit Fields[1]

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

NOTES:

1. This table also represents VEX restrictions.

Table 2-42 lists the #UD conditions of instruction encoding of opmask register using EVEX.aaa and EVEX.z

Table 2-42. #UD Conditions of Opmask Related Encoding Field

Notation	Position	Operand Encoding	64-bit #UD	Non-64-bit #UD
EVEX.aaa	P[18:16]	Instructions do not use opmask for conditional processing[1].	If aaa != 000b	If aaa != 000b
		Opmask used as conditional processing mask and updated at completion[2].	If aaa = 000b	If aaa = 000b;
		Opmask used as conditional processing.	None (valid[3])	None (valid[1])
EVEX.z	P[23]	Vector instruction using opmask as source or destination[4].	If EVEX.z != 0	If EVEX.z != 0
		Store instructions or gather/scatter instructions.	If EVEX.z != 0	If EVEX.z != 0
		Instructions with EVEX.aaa = 000b.	If EVEX.z != 0	If EVEX.z != 0
VEX.vvvv	Varies	K-regs are instruction operands not mask control.	If vvvv = 0xxxb	None

NOTES:

1. E.g., VPBROADCASTMxxx, VPMOVM2x, VPMOVx2M.
2. E.g., Gather/Scatter family.

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

Table 2-43 lists the #UD conditions of EVEX bit fields that depends on the context of EVEX.b.

Table 2-43. #UD Conditions Dependent on EVEX.b Context

Notation	Position	Operand Encoding	64-bit #UD	Non-64-bit #UD
EVEX.L'Lb	P[22 : 20]	Reg-reg, FP instructions with rounding semantic.	None (valid[1])	None (valid[1])
		Other reg-reg, FP instructions that can cause #XM.	None (valid[2])	None (valid[2])
		Other reg-mem instructions in Table 2-36.	None (valid[3])	None (valid[3])
		Other instruction classes[4] in Table 2-37.	If EVEX.b = 1	If EVEX.b = 1

NOTES:

1. L'L specifies rounding control, see Table 2-38, supports {er} syntax.
2. L'L is ignored.
3. L'L specifies vector length, see Table 2-38, supports embedded broadcast syntax
4. L'L specifies either vector length or ignored.

2.7.12 Device Not Available

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

2.7.13 Scalar Instructions

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

2.8 EXCEPTION CLASSIFICATIONS OF EVEX-ENCODED INSTRUCTIONS

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

Table 2-44. EVEX-Encoded Instruction Exception Class Summary

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

Table 2-45 lists EVEX-encoded instruction mnemonic by exception classes.

Table 2-45. EVEX Instructions in Each Exception Class

Exception Class	Instruction
Type E1	VMOVAPD, VMOVAPS, VMOVDQA32, VMOVDQA64
Type E1NF	VMOVNTDQ, VMOVNTDQA, VMOVNTPD, VMOVNTPS
Type E2	VADDPD, VADDPH, VADDPS, VCMPPD, VCMPPH, VCMPPS, VCVTDQ2PH, VCVTDQ2PS, VCVTPD2DQ, VCVTPD2PH, VCVTPD2PS, VCVTPD2QQ, VCVTPD2UQQ, VCVTPD2UDQ, VCVTPH2DQ, VCVTPH2PD, VCVTPH2QQ, VCVTPH2UDQ, VCVTPH2UQQ, VCVTPH2UW, VCVTPH2W, VCVTPS2DQ, VCVTPS2PD, VCVTPS2QQ, VCVTPS2UDQS, VCVTPS2UQQ, VCVTQQ2PD, VCVTQQ2PH, VCVTQQ2PS, VCVTTPD2DQ, VCVTTPD2QQ, VCVTTPD2UDQ, VCVTTPD2UQQ, VCVTTPH2DQ, VCVTTPH2QQ, VCVTTPH2UDQ, VCVTTPH2UQQ, VCVTTPH2UW, VCVTTPH2W, VCVTTPS2DQ, VCVTTPS2QQ, VCVTTPS2UDQ, VCVTTPS2UQQ, VCVTUDQ2PH, VCVTUDQ2PS, VCVTUQQ2PD, VCVTUQQ2PH, VCVTUQQ2PS, VCVTUW2PH, VCVTW2PH, VDIVPD, VDIVPH, VDIVPS, VEXP2PD, VEXP2PS, VFIXUPIMMPD, VFIXUPIMMPS, VFMADDxxxPD, VFMADDxxxPH, VFMADDxxxPS, VFMADDSUBxxxPD, VFMADDSUBxxxPH, VFMADDSUBxxxPS, VFMSUBADDxxxPD, VFMSUBADDxxxPH, VFMSUBADDxxxPS, VFMSUBxxxPD, VFMSUBxxxPH, VFMSUBxxxPS, VFNMADDxxxPD, VFNMADDxxxPH, VFNMADDxxxPS, VFNMSUBxxxPD, VFNMSUBxxxPH, VFNMSUBxxxPS, VGETEXPPD, VGETEXPPH, VGETEXPPS, VGETMANTPD, VGETMANTPH, VGETMANTPS, VGETMANTSH, VMAXPD, VMAXPH, VMAXPS, VMINPD, VMINPH, VMINPS, VMULPD, VMULPH, VMULPS, VRANGEPD, VRANGEPS, VREDUCEPD, VREDUCEPH, VREDUCEPS, VRNDSCALEPD, VRNDSCALEPH, VRNDSCALEPS, VRCP28PD, VRCP28PS, VRSQRT28PD, VRSQRT28PS, VSCALEFPD, VSCALEFPS, VSQRTPD, VSQRTPH, VSQRTPS, VSUBPD, VSUBPH, VSUBPS

Table 2-45. EVEX Instructions in Each Exception Class (Contd.)

Exception Class	Instruction
Type E3	VADDSD, VADDSH, VADDSS, VCMPSD, VCMPSH, VCMPSS, VCVTSD2SH, VCVTSD2SS, VCVTSH2SD, VCVTSH2SS, VCVTSS2SD, VCVTSS2SH, VDIVSD, VDIVSH, VDIVSS, VFMADDxxxSD, VFMADDxxxSH, VFMADDxxxSS, VFMSUBxxxSD, VFMSUBxxxSH, VFMSUBxxxSS, VFNMADDxxxSD, VFNMADDxxxSH, VFNMADDxxxSS, VFNMSUBxxxSD, VFNMSUBxxxSH, VFNMSUBxxxSS, VFIXUPIMMSD, VFIXUPIMMSS, VGETEXPSD, VGETEXPSH, VGETEXPSS, VGETMANTSD, VGETMANTSH, VGETMANTSS, VMAXSD, VMAXSH, VMAXSS, VMINSD, VMINSH, VMINSS, VMULSD, VMULSH, VMULSS, VRANGESD, VRANGESS, VREDUCESD, VREDUCESH, VREDUCESS, VRNDSCALESD, VRNDSCALESH, VRNDSCALESS, VSCALEFSD, VSCALEFSH, VSCALEFSS, VRCP28SD, VRCP28SS, VRSQRT28SD, VRSQRT28SS, VSQRTSD, VSQRTSH, VSQRTSS, VSUBSD, VSUBSH, VSUBSS
Type E3NF	VCOMISD, VCOMISH, VCOMISS, VCVTSD2SI, VCVTSD2USI, VCVTSH2SI, VCVTSH2USI, VCVTSI2SD, VCVTSI2SH, VCVTSI2SS, VCVTSS2SI, VCVTSS2USI, VCVTTSD2SI, VCVTTSD2USI, VCVTTSH2SI, VCVTTSH2USI, VCVTTSS2SI, VCVTTSS2USI, VCVTUSI2SD, VCVTUSI2SH, VCVTUSI2SS, VUCOMISD, VUCOMISH, VUCOMISS
Type E4	VANDPD, VANDPS, VANDNPD, VANDNPS, VBLENDMPD, VBLENDMPS, VFCMADDCPH, VFCMULCPH, VFMADDCPH, VFMULCPH, VFPCLASSPD, VFPCLASSPH, VFPCLASSPS, VORPD, VORPS, VPABSD, VPABSQ, VPADDD, VPADDQ, VPANDD, VPANDQ, VPANDND, VPANDNQ, VPBLENDMB, VPBLENDMD, VPBLENDMQ, VPBLENDMW, VPCMPD, VPCMPEQD, VPCMPEQQ, VPCMPGTD, VPCMPGTQ, VPCMPQ, VPCMPUD, VPCMPUQ, VPLZCNTD, VPLZCNTQ, VPMADD52LUQ, VPMADD52HUQ, VPMAXSD, VPMAXSQ, VPMAXUD, VPMAXUQ, VPMINSD, VPMINSQ, VPMINUD, VPMINUQ, VPMULLD, VPMULLQ, VPMULUDQ, VPMULDQ, VPORD, VPORQ, VPROLD, VPROLQ, VPROLVD, VPROLVQ, VPRORD, VPRORQ, VPRORVD, VPRORVQ, VPSHUFBITQMB, (VPSLLD, VPSLLQ, VPSRAD, VPSRAQ, VPSRAVW, VPSRAVW, VPSRAVD, VPSRAVQ, VPSRLD, VPSRLQ)[1], VPSUBD, VPSUBQ, VPSUBUSB, VPSUBUSW, VPTERNLOGD, VPTERNLOGQ, VPTESTMD, VPTESTMQ, VPTESTNMD, VPTESTNMQ, VPXORD, VPXORQ, VPSLLVD, VPSLLVQ, VRCP14PD, VRCP14PS, VRCPPH, VRSQRT14PD, VRSQRT14PS, VRSQRTPH, VXORPD, VXORPS
E4.nb[2]	VCOMPRESSPD, VCOMPRESSPS, VEXPANDPD, VEXPANDPS, VMOVDQU8, VMOVDQU16, VMOVDQU32, VMOVDQU64, VMOVUPD, VMOVUPS, VPABSB, VPABSW, VPADDB, VPADDW, VPADDSB, VPADDSW, VPADDUSB, VPADDUSW, VPAVGB, VPAVGW, VPCMPB, VPCMPEQB, VPCMPEQW, VPCMPGTB, VPCMPGTW, VPCMPW, VPCMPUB, VPCMPUW, VPCOMPRESSD, VPCOMPRESSQ, VPEXPANDD, VPEXPANDQ, VPMAXSB, VPMAXSW, VPMAXUB, VPMAXUW, VPMINSB, VPMINSW, VPMINUB, VPMINUW, VPMULHRSW, VPMULHUW, VPMULHW, VPMULLW, VPSLLVW, VPSLLW, VPSRAW, VPSRLVW, VPSRLW, VPSUBB, VPSUBW, VPSUBSB, VPSUBSW, VPTESTMB, VPTESTMW, VPTESTNMB, VPTESTNMW
Type E4NF	VALIGND, VALIGNQ, VPACKSSDW, VPACKUSDW, VPCONFLICTD, VPCONFLICTQ, VPERMD, VPERMI2D, VPERMI2PS, VPERMI2PD, VPERMI2Q, VPERMPD, VPERMPS, VPERMQ, VPERMT2D, VPERMT2PS, VPERMT2Q, VPERMT2PD, VPERMILPD, VPERMILPS, VPMULTISHIFTQB, VPSHUFD, VPUNPCKHDQ, VPUNPCKHQDQ, VPUNPCKLDQ, VPUNPCKLQDQ, VSHUFF32X4, VSHUFF64X2, VSHUFI32X4, VSHUFI64X2, VSHUFPD, VSHUFPS, VUNPCKHPD, VUNPCKHPS, VUNPCKLPD, VUNPCKLPS
E4NF.nb[2]	VDBPSADBW, VPACKSSWB, VPACKUSWB, VPALIGNR, VPMADDWD, VPMADDUBSW, VMOVSHDUP, VMOVSLDUP, VPSADBW, VPSHUFB, VPSHUFHW, VPSHUFLW, VPSLLDQ, VPSRLDQ, VPSLLW, VPSRAW, VPSRLW, (VPSLLD, VPSLLQ, VPSRAD, VPSRAQ, VPSRLD, VPSRLQ)[3], VPUNPCKHBW, VPUNPCKHWD, VPUNPCKLBW, VPUNPCKLWD, VPERMW, VPERMI2W, VPERMT2W
Type E5	PMOVSXBW, PMOVSXBW, PMOVSXBD, PMOVSXBQ, PMOVSXWD, PMOVSXWQ, PMOVSXDQ, PMOVZXBW, PMOVZXBD, PMOVZXBQ, PMOVZXWD, PMOVZXWQ, PMOVZXDQ, VCVTDQ2PD, VCVTUDQ2PD, VMOVSH, VPMOVSXxx, VPMOVZXxx,
Type E5NF	VMOVDDUP
Type E6	VBROADCASTF32X2, VBROADCASTF32X4, VBROADCASTF64X2, VBROADCASTF32X8, VBROADCASTF64X4, VBROADCASTI32X2, VBROADCASTI32X4, VBROADCASTI64X2, VBROADCASTI32X8, VBROADCASTI64X4, VBROADCASTSD, VBROADCASTSS, VFPCLASSSD, VFPCLASSSS, VPBROADCASTB, VPBROADCASTD, VPBROADCASTW, VPBROADCASTQ, VPMOVQB, VPMOVSQB, VPMOVUSQB, VPMOVQW, VPMOVSQW, VPMOVUSQW, VPMOVQD, VPMOVSQD, VPMOVUSQD, VPMOVDB, VPMOVSDB, VPMOVUSDB, VPMOVDW, VPMOVSDW, VPMOVUSDW, VPMOVWB, VPMOVSWB, VPMOVUSWB
Type E6NF	VEXTRACTF32X4, VEXTRACTF32X8, VEXTRACTF64X2, VEXTRACTF64X4, VEXTRACTI32X4, VEXTRACTI32X8, VEXTRACTI64X2, VEXTRACTI64X4, VINSERTF32X4, VINSERTF32X8, VINSERTF64X2, VINSERTF64X4, VINSERTI32X4, VINSERTI32X8, VINSERTI64X2, VINSERTI64X4, VPBROADCASTMB2Q, VPBROADCASTMW2D

Table 2-45. EVEX Instructions in Each Exception Class (Contd.)

Exception Class	Instruction
Type E7NM.128[4]	VMOVHLPS, VMOVLHPS
Type E7NM.	(VPBROADCASTD, VPBROADCASTQ, VPBROADCASTB, VPBROADCASTW)[5], VPMOVB2M, VPMOVD2M, VPMOVM2B, VPMOVM2D, VPMOVM2Q, VPMOVM2W, VPMOVQ2M, VPMOVW2M
Type E9NF	VEXTRACTPS, VINSERTPS, VMOVHPD, VMOVHPS, VMOVLPD, VMOVLPS, VMOVD, VMOVQ, VMOVW, VPEXTRB, VPEXTRD, VPEXTRW, VPEXTRQ, VPINSRB, VPINSRD, VPINSRW, VPINSRQ
Type E10	VFCMADDCSH, VFMADDCSH, VFCMULCSH, VFMULCSH, VFPCLASSSH, VMOVSD, VMOVSS, VRCP14SD, VRCP14SS, VRCPSH, VRSQRT14SD, VRSQRT14SS, VRSQRTSH
Type E10NF	(VCVTSI2SD, VCVTUSI2SD)[6]
Type E11	VCVTPH2PS, VCVTPS2PH
Type E12	VGATHERDPS, VGATHERDPD, VGATHERQPS, VGATHERQPD, VPGATHERDD, VPGATHERDQ, VPGATHERQD, VPGATHERQQ, VPSCATTERDD, VPSCATTERDQ, VPSCATTERQD, VPSCATTERQQ, VSCATTERDPD, VSCATTERDPS, VSCATTERQPD, VSCATTERQPS
Type E12NP	VGATHERPF0DPD, VGATHERPF0DPS, VGATHERPF0QPD, VGATHERPF0QPS, VGATHERPF1DPD, VGATHERPF1DPS, VGATHERPF1QPD, VGATHERPF1QPS, VSCATTERPF0DPD, VSCATTERPF0DPS, VSCATTERPF0QPD, VSCATTERPF0QPS, VSCATTERPF1DPD, VSCATTERPF1DPS, VSCATTERPF1QPD, VSCATTERPF1QPS

NOTES:

1. Operand encoding Full tupletype with immediate.
2. Embedded broadcast is not supported with the ".nb" suffix.
3. Operand encoding Mem128 tuple type.
4. #UD raised if EVEX.L'L !=00b (VL=128).
5. The source operand is a general purpose register.
6. W0 encoding only.

2.8.1 Exceptions Type E1 and E1NF of EVEX-Encoded Instructions

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

Table 2-46. Type E1 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43. ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X	X	EVEX.512: Memory operand is not 64-byte aligned. EVEX.256: Memory operand is not 32-byte aligned. EVEX.128: Memory operand is not 16-byte aligned.
			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.

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

Table 2-47. Type E1NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43. ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X	X	EVEX.512: Memory operand is not 64-byte aligned. EVEX.256: Memory operand is not 32-byte aligned. EVEX.128: Memory operand is not 16-byte aligned.
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.

2.8.2 Exceptions Type E2 of EVEX-Encoded Instructions

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

Table 2-48. Type E2 Class Exception Conditions

Exception	Real	Virtual 8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception, {sae} or {er} not set, and CR4.OSXMMEXCPT[bit 10] = 1.

2.8.3 Exceptions Type E3 and E3NF of EVEX-Encoded Instructions

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

Table 2-49. Type E3 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception, {sae} or {er} not set, and CR4.OSX-MMEXCPT[bit 10] = 1.

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

Table 2-50. Type E3NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			EVEX prefix.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception, {sae} or {er} not set, and CR4.OSXMMEXCPT[bit 10] = 1.

2.8.4 Exceptions Type E4 and E4NF of EVEX-Encoded Instructions

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

Table 2-51. Type E4 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43 and in E4.nb subclass (see E4.nb entries in Table 2-45). ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

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

Table 2-52. Type E4NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43 and in E4NF.nb subclass (see E4NF.nb entries in Table 2-45). ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.

2.8.5 Exceptions Type E5 and E5NF

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

Table 2-53. Type E5 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43. ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

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

Table 2-54. Type E5NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43. ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

2.8.6 Exceptions Type E6 and E6NF

Table 2-55. Type E6 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43. ▪ Instruction specific EVEX.L'L restriction not met.
			X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
			X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
Page Fault #PF(fault-code)			X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)			X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

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

Table 2-56. Type E6NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43. ▪ Instruction specific EVEX.L'L restriction not met.
			X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
			X	X	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 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

2.8.7 Exceptions Type E7NM

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

Table 2-57. Type E7NM Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43. ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1.

2.8.8 Exceptions Type E9 and E9NF

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

Table 2-58. Type E9 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43. ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.

Table 2-58. Type E9 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

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

Table 2-59. Type E9NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43. ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

2.8.9 Exceptions Type E10 and E10NF

EVEX-encoded scalar instructions that ignore EVEX.L'L vector length encoding, do not cause a SIMD FP exception, and support memory fault suppression follow exception class E10.

Table 2-60. Type E10 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

EVEX-encoded scalar instructions that ignore EVEX.L'L vector length encoding, do not cause a SIMD FP exception, and do not support memory fault suppression follow exception class E10NF.

Table 2-61. Type E10NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

2.8.10 Exceptions Type E11 (EVEX-only, Mem Arg, No AC, Floating-point Exceptions)

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

Table 2-62. Type E11 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43. ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF (fault-code)		X	X	X	If fault suppression not set, and a page fault.
SIMD Floating-Point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception, {sae} not set, and CR4.OSXMMEX-CPT[bit 10] = 1.

2.8.11 Exceptions Type E12 and E12NP (VSIB Mem Arg, No AC, No Floating-point Exceptions)

Table 2-63. Type E12 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43. ▪ Instruction specific EVEX.L'L restriction not met. ▪ If vvvv != 1111b.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	NA	If address size attribute is 16 bit.
	X	X	X	X	If ModR/M.mod = '11b'.
	X	X	X	X	If ModR/M.rm != '100b'.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
	X	X	X	X	If k0 is used (gather or scatter operation).
	X	X	X	X	If index = destination register (gather operation).
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF (fault-code)		X	X	X	For a page fault.

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

Table 2-64. Type E12NP Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43. ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	NA	If address size attribute is 16 bit.
	X	X	X	X	If ModR/M.mod = '11b'.
	X	X	X	X	If ModR/M.rm != '100b'.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
	X	X	X	X	If k0 is used (gather or scatter operation).
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.

2.9 EXCEPTION CLASSIFICATIONS OF OPMASK INSTRUCTIONS, TYPE K20 AND TYPE K21

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

2.9.1 Exceptions Type K20

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

Table 2-65. TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If relevant CPUID feature flag is '0'.
	X	X			If a VEX prefix is present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41.
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
			X	X	If ModRM:[7:6] != 11b.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.

2.9.2 Exceptions Type K21

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

Table 2-66. TYPE K21 Exception Definition (VEX-Encoded OpMask Instructions Addressing Memory)

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If relevant CPUID feature flag is '0'.
	X	X			If a VEX prefix is present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.

Table 2-66. TYPE K21 Exception Definition (VEX-Encoded OpMask Instructions Addressing Memory)

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
			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	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

2.10 INTEL® AMX INSTRUCTION EXCEPTION CLASSES

Alignment exceptions: The Intel AMX instructions that access memory will never generate #AC exceptions.

Table 2-67. Intel® AMX Exception Classes

Class	Description
AMX-E1	• #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes. • #UD if CR4.OSXSAVE ≠ 1. • #UD if XCR0[18:17] ≠ 0b11. • #UD if IA32_EFER.LMA ≠ 1 OR CS.L ≠ 1. • #UD if VVVV ≠ 0b1111.
	• #GP based on palette and configuration checks (see pseudocode). • #GP if the memory address is in a non-canonical form.
	• #SS(0) if the memory address referencing the SS segment is in a non-canonical form.
	• #PF if a page fault occurs.
AMX-E2	• #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes. • #UD if CR4.OSXSAVE ≠ 1. • #UD if XCR0[18:17] ≠ 0b11. • #UD if IA32_EFER.LMA ≠ 1 OR CS.L ≠ 1. • #UD if VVVV ≠ 0b1111.
	• #GP if the memory address is in a non-canonical form.
	• #SS(0) if the memory address referencing the SS segment is in a non-canonical form.
	• #PF if a page fault occurs.

Table 2-67. Intel® AMX Exception Classes (Contd.)

Class	Description
AMX-E3	• #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes. • #UD if CR4.OSXSAVE ≠ 1. • #UD if XCR0[18:17] ≠ 0b11. • #UD if IA32_EFER.LMA ≠ 1 OR CS.L ≠ 1. • #UD if VVVV ≠ 0b1111. • #UD if not using SIB addressing. • #UD if TILES_CONFIGURED == 0. • #UD if tsrc or tdest are not valid tiles. • #UD if tsrc/tdest are ≥ palette_table[tilecfg.palette_id].max_names. • #UD if tsrc.colbytes mod 4 ≠ 0 OR tdest.colbytes mod 4 ≠ 0. • #UD if tilecfg.start_row ≥ tsrc.rows OR tilecfg.start_row ≥ tdest.rows.
	• #GP if the memory address is in a non-canonical form.
	• #SS(0) if the memory address referencing the SS segment is in a non-canonical form.
	• #PF if any memory operand causes a page fault.
	• #NM if XFD[18] == 1.
AMX-E4	• #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes. • #UD if CR4.OSXSAVE ≠ 1. • #UD if XCR0[18:17] ≠ 0b11. • #UD if IA32_EFER.LMA ≠ 1 OR CS.L ≠ 1. • #UD if srcdest == src1 OR src1 == src2 OR srcdest == src2. • #UD if TILES_CONFIGURED == 0. • #UD if srcdest.colbytes mod 4 ≠ 0. • #UD if src1.colbytes mod 4 ≠ 0. • #UD if src2.colbytes mod 4 ≠ 0. • #UD if srcdest/src1/src2 are not valid tiles. • #UD if srcdest/src1/src2 are ≥ palette_table[tilecfg.palette_id].max_names. • #UD if srcdest.colbytes ≠ src2.colbytes. • #UD if srcdest.rows ≠ src1.rows. • #UD if src1.colbytes / 4 ≠ src2.rows. • #UD if srcdest.colbytes > tmul_maxn. • #UD if src2.colbytes > tmul_maxn. • #UD if src1.colbytes/4 > tmul_maxk. • #UD if src2.rows > tmul_maxk.
	• #NM if XFD[18] == 1.
AMX-E5	• #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes. • #UD if CR4.OSXSAVE ≠ 1. • #UD if XCR0[18:17] ≠ 0b11. • #UD if IA32_EFER.LMA ≠ 1 OR CS.L ≠ 1. • #UD if VVVV ≠ 0b1111. • #UD if TILES_CONFIGURED == 0. • #UD if tdest is not a valid tile. • #UD if tdest is ≥ palette_table[tilecfg.palette_id].max_names.
	• #NM if XFD[18] == 1.
AMX-E6	• #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes. • #UD if CR4.OSXSAVE ≠ 1. • #UD if XCR0[18:17] ≠ 0b11. • #UD if IA32_EFER.LMA ≠ 1 OR CS.L ≠ 1. • #UD if VVVV ≠ 0b1111.

CHAPTER 3
INSTRUCTION SET REFERENCE, A-L

This chapter describes the instruction set for the Intel 64 and IA-32 architectures (A-L) in IA-32e, protected, virtual-8086, and real-address modes of operation. The set includes general-purpose, x87 FPU, MMX, SSE/SSE2/SSE3/SSSE3/SSE4, AESNI/PCLMULQDQ, AVX, and system instructions. See also Chapter 4, "Instruction Set Reference, M-U," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B; Chapter 5, "Instruction Set Reference, V," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C; and Chapter 6, "Instruction Set Reference, W-Z," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D.

For each instruction, each operand combination is described. A description of the instruction and its operand, an operational description, a description of the effect of the instructions on flags in the EFLAGS register, and a summary of exceptions that can be generated are also provided.

3.1 INTERPRETING THE 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.

3.1.1 Instruction Format

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

CMC—Complement Carry Flag [this is an example]

Opcode	Instruction	Op/En	64/32-bit Mode	CPUID Feature Flag	Description
F5	CMC	ZO	V/V	N/A	Complement carry flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

3.1.1.1 Opcode Column in the Instruction Summary Table (Instructions without VEX Prefix)

The "Opcode" column in the table above shows the object code produced for each form of the instruction. When possible, codes are given as hexadecimal bytes in the same order in which they appear in memory. Definitions of entries other than hexadecimal bytes are as follows:

- **NP** — Indicates the use of 66/F2/F3 prefixes (beyond those already part of the instructions opcode) are not allowed with the instruction. Such use will either cause an invalid-opcode exception (#UD) or result in the encoding for a different instruction.
- **NFx** — Indicates the use of F2/F3 prefixes (beyond those already part of the instructions opcode) are not allowed with the instruction. Such use will either cause an invalid-opcode exception (#UD) or result in the encoding for a different instruction.
- **REX.W** — Indicates the use of a REX prefix that affects operand size or instruction semantics. The ordering of the REX prefix and other optional/mandatory instruction prefixes are discussed Chapter 2. Note that REX prefixes that promote legacy instructions to 64-bit behavior are not listed explicitly in the opcode column.
- **/digit** — A digit between 0 and 7 indicates that the ModR/M byte of the instruction uses only the r/m (register or memory) operand. The reg field contains the digit that provides an extension to the instruction's opcode.
- **/r —** Indicates that the ModR/M byte of the instruction contains a register operand and an r/m operand.
- **cb, cw, cd, cp, co, ct —** A 1-byte (cb), 2-byte (cw), 4-byte (cd), 6-byte (cp), 8-byte (co) or 10-byte (ct) value following the opcode. This value is used to specify a code offset and possibly a new value for the code segment register.
- **ib, iw, id, io —** A 1-byte (ib), 2-byte (iw), 4-byte (id) or 8-byte (io) immediate operand to the instruction that follows the opcode, ModR/M bytes or scale-indexing bytes. The opcode determines if the operand is a signed value. All words, doublewords, and quadwords are given with the low-order byte first.
- **+rb, +rw, +rd, +ro —** Indicated the lower 3 bits of the opcode byte is used to encode the register operand without a modR/M byte. The instruction lists the corresponding hexadecimal value of the opcode byte with low 3 bits as 000b. In non-64-bit mode, a register code, from 0 through 7, is added to the hexadecimal value of the opcode byte. In 64-bit mode, indicates the four bit field of REX.b and opcode[2:0] field encodes the register operand of the instruction. "+ro" is applicable only in 64-bit mode. See Table 3-1 for the codes.
- +**i —** A number used in floating-point instructions when one of the operands is ST(i) from the FPU register stack. The number i (which can range from 0 to 7) is added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte.

Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro

byte register			word register			dword register			quadword register (64-Bit Mode only)		
Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field
AL	None	0	AX	None	0	EAX	None	0	RAX	None	0
CL	None	1	CX	None	1	ECX	None	1	RCX	None	1
DL	None	2	DX	None	2	EDX	None	2	RDX	None	2
BL	None	3	BX	None	3	EBX	None	3	RBX	None	3
AH	Not encodable (N.E.)	4	SP	None	4	ESP	None	4	N/A	N/A	N/A
CH	N.E.	5	BP	None	5	EBP	None	5	N/A	N/A	N/A
DH	N.E.	6	SI	None	6	ESI	None	6	N/A	N/A	N/A
BH	N.E.	7	DI	None	7	EDI	None	7	N/A	N/A	N/A
SPL	Yes	4	SP	None	4	ESP	None	4	RSP	None	4
BPL	Yes	5	BP	None	5	EBP	None	5	RBP	None	5

Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro (Contd.)

byte register			word register			dword register			quadword register (64-Bit Mode only)		
Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field
SIL	Yes	6	SI	None	6	ESI	None	6	RSI	None	6
DIL	Yes	7	DI	None	7	EDI	None	7	RDI	None	7
Registers R8 - R15 (see below): Available in 64-Bit Mode Only											
R8B	Yes	0	R8W	Yes	0	R8D	Yes	0	R8	Yes	0
R9B	Yes	1	R9W	Yes	1	R9D	Yes	1	R9	Yes	1
R10B	Yes	2	R10W	Yes	2	R10D	Yes	2	R10	Yes	2
R11B	Yes	3	R11W	Yes	3	R11D	Yes	3	R11	Yes	3
R12B	Yes	4	R12W	Yes	4	R12D	Yes	4	R12	Yes	4
R13B	Yes	5	R13W	Yes	5	R13D	Yes	5	R13	Yes	5
R14B	Yes	6	R14W	Yes	6	R14D	Yes	6	R14	Yes	6
R15B	Yes	7	R15W	Yes	7	R15D	Yes	7	R15	Yes	7

3.1.1.2 Opcode Column in the Instruction Summary Table (Instructions with VEX prefix)

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

- **VEX** — Indicates the presence of the VEX prefix is required. The VEX prefix can be encoded using the three-byte form (the first byte is C4H), or using the two-byte form (the first byte is C5H). The two-byte form of VEX only applies to those instructions that do not require the following fields to be encoded: VEX.mmmmm, VEX.W, VEX.X, VEX.B. Refer to Section 2.3 for more detail on the VEX prefix.

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

  — **128,256:** VEX.L field can be 0 (denoted by VEX.128, VEX.L0, or VEX.LZ) or 1 (denoted by VEX.256 or VEX.L1). 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.L0 or VEX.L1 is present in the opcode column: The specified VEX.L value is required for encoding this instruction but does not have the connotation of specifying vector length.
  - 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 values map 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 2.3 on the subfield definitions within VEX.
  - **WIG:** can use C5H form (if not requiring VEX.mmmmm) or VEX.W value is ignored in the C4H form of VEX prefix.
  - If WIG is present, the instruction may be encoded using either the two-byte form or the three-byte form of VEX. When encoding the instruction using the three-byte form of VEX, the value of VEX.W is ignored.
- **opcode** — Instruction opcode.
- **/is4** — An 8-bit immediate byte is present containing a source register specifier in either imm8[7:4] (for 64-bit mode) or imm8[6:4] (for 32-bit mode), and instruction-specific payload in imm8[3:0].
- In general, the encoding o f VEX.R, VEX.X, VEX.B field are not shown explicitly in the opcode column. The encoding scheme of VEX.R, VEX.X, VEX.B fields must follow the rules defined in Section 2.3.

EVEX.[128,256,512,LLIG].[66,F2,F3].0F/0F3A/0F38.[W0,W1,WIG] opcode [/r] [/ib]

- **EVEX** — The EVEX prefix is encoded using the four-byte form (the first byte is 62H). Refer to Section 2.7.1 for more detail on the EVEX prefix.

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

  - **128, 256, 512, LLIG:** This corresponds to the vector length; three values are allowed by EVEX: 512-bit, 256-bit and 128-bit. Alternatively, vector length is ignored (LIG) for certain instructions; this typically applies to scalar instructions operating on one data element of a vector register.
  - **66,F2,F3:** The presence of these value maps to the EVEX.pp field encodings. The corresponding VEX.pp value affects the "opcode" byte in the same way as if a SIMD prefix (66H, F2H or F3H) does to the ensuing opcode byte. Thus a non-zero encoding of VEX.pp may be considered as an implied 66H/F2H/F3H prefix.
  - **0F,0F3A,0F38:** The presence maps to a valid encoding of the EVEX.mmm field. Only three encoded values of EVEX.mmm are defined as valid, corresponding to the escape byte sequence of 0FH, 0F3AH, and 0F38H. The effect of a valid EVEX.mmm encoding on the ensuing opcode byte is the same as if the corresponding escape byte sequence on the ensuing opcode byte for non-EVEX encoded instructions. Thus a valid encoding of EVEX.mmm may be considered as an implied escape byte sequence of either 0FH, 0F3AH or 0F38H.
  - **W0:** EVEX.W=0.

  - **W1:** EVEX.W=1.
  - **WIG:** EVEX.W bit ignored
- **opcode** — Instruction opcode.
- In general, the encoding of EVEX.R and R', EVEX.X and X', and EVEX.B and B' fields are not shown explicitly in the opcode column.

NOTE

Previously, the terms NDS, NDD, and DDS were used in instructions with an EVEX (or VEX) prefix. These terms indicated that the vvvv field was valid for encoding, and specified register usage. These terms are no longer necessary and are redundant with the instruction operand encoding tables provided with each instruction. The instruction operand encoding tables give explicit details on all operands, indicating where every operand is stored and if they are read or written. If vvvv is not listed as an operand in the instruction operand encoding table, then EVEX (or VEX) vvvv must be 0b1111.

3.1.1.3 Instruction Column in the Opcode Summary Table

The "Instruction" column gives the syntax of the instruction statement as it would appear in an ASM386 program. The following is a list of the symbols used to represent operands in the instruction statements:

- **rel8** — A relative address in the range from 128 bytes before the end of the instruction to 127 bytes after the end of the instruction.
- **rel16, rel32** — A relative address within the same code segment as the instruction assembled. The rel16 symbol applies to instructions with an operand-size attribute of 16 bits; the rel32 symbol applies to instructions with an operand-size attribute of 32 bits.
- **ptr16:16, ptr16:32** — A far pointer, typically to a code segment different from that of the instruction. The notation *16:16* indicates that the value of the pointer has two parts. The value to the left of the colon is a 16-bit selector or value destined for the code segment register. The value to the right corresponds to the offset within the destination segment. The ptr16:16 symbol is used when the instruction's operand-size attribute is 16 bits; the ptr16:32 symbol is used when the operand-size attribute is 32 bits.
- **r8** — One of the byte general-purpose registers: AL, CL, DL, BL, AH, CH, DH, BH, BPL, SPL, DIL, and SIL; or one of the byte registers (R8B - R15B) available when using REX.R and 64-bit mode.
- **r16** — One of the word general-purpose registers: AX, CX, DX, BX, SP, BP, SI, DI; or one of the word registers (R8-R15) available when using REX.R and 64-bit mode.
- **r32** — One of the doubleword general-purpose registers: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI; or one of the doubleword registers (R8D - R15D) available when using REX.R in 64-bit mode.
- **r64** — One of the quadword general-purpose registers: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8–R15. These are available when using REX.R and 64-bit mode.
- **imm8** — An immediate byte value. The imm8 symbol can be a signed number between –128 and +127 inclusive; an unsigned number between 0 and 255 inclusive; or a bitmap when an instruction uses its individual bits. For instructions in which imm8 is combined with a word or doubleword operand, the immediate value is sign-extended to form a word or doubleword. The upper byte of the word is filled with the topmost bit of the immediate value.
- **imm16** — An immediate word value used for instructions whose operand-size attribute is 16 bits. This is a number between –32,768 and +32,767 inclusive.
- **imm32** — An immediate doubleword value used for instructions whose operand-size attribute is 32 bits. It allows the use of a number between +2,147,483,647 and –2,147,483,648 inclusive.
- **imm64** — An immediate quadword value used for instructions whose operand-size attribute is 64 bits. The value allows the use of a number between +9,223,372,036,854,775,807 and –9,223,372,036,854,775,808 inclusive.
- **/ib** — A single-byte value.

- **r/m8** — A byte operand that is either the contents of a byte general-purpose register (AL, CL, DL, BL, AH, CH, DH, BH, BPL, SPL, DIL, and SIL) or a byte from memory. Byte registers R8B - R15B are available using REX.R in 64-bit mode.
- **r/m16** — A word general-purpose register or memory operand used for instructions whose operand-size attribute is 16 bits. The word general-purpose registers are: AX, CX, DX, BX, SP, BP, SI, DI. The contents of memory are found at the address provided by the effective address computation. Word registers R8W - R15W are available using REX.R in 64-bit mode.
- **r/m32** — A doubleword general-purpose register or memory operand used for instructions whose operand-size attribute is 32 bits. The doubleword general-purpose registers are: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI. The contents of memory are found at the address provided by the effective address computation. Doubleword registers R8D - R15D are available when using REX.R in 64-bit mode.
- **r/m64** — A quadword general-purpose register or memory operand used for instructions whose operand-size attribute is 64 bits when using REX.W. Quadword general-purpose registers are: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8–R15; these are available only in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.
- **reg** — A general-purpose register used for instructions when the width of the register does not matter to the semantics of the operation of the instruction. The register can be r16, r32, or r64.
- **m** — A 16-, 32- or 64-bit operand in memory.
- **m8** — A byte operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. In 64-bit mode, it is pointed to by the RSI or RDI registers.
- **m16** — A word operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions.
- **m32** — A doubleword operand in memory. The contents of memory are found at the address provided by the effective address computation.
- **m64** — A memory quadword operand in memory.
- **m128** — A memory double quadword operand in memory.
- **m16:16, m16:32 & m16:64** — A memory operand containing a far pointer composed of two numbers. The number to the left of the colon corresponds to the pointer's segment selector. The number to the right corresponds to its offset.
- **m16&32, m16&16, m32&32, m16&64** — A memory operand consisting of data item pairs whose sizes are indicated on the left and the right side of the ampersand. All memory addressing modes are allowed. The m16&16 and m32&32 operands are used by the BOUND instruction to provide an operand containing an upper and lower bounds for array indices. The m16&32 operand is used by LIDT and LGDT to provide a word with which to load the limit field, and a doubleword with which to load the base field of the corresponding GDTR and IDTR registers. The m16&64 operand is used by LIDT and LGDT in 64-bit mode to provide a word with which to load the limit field, and a quadword with which to load the base field of the corresponding GDTR and IDTR registers.
- **m80bcd**— A Binary Coded Decimal (BCD) operand in memory, 80 bits.
- **moffs8, moffs16, moffs32, moffs64** — A simple memory variable (memory offset) of type byte, word, or doubleword used by some variants of the MOV instruction. The actual address is given by a simple offset relative to the segment base. No ModR/M byte is used in the instruction. The number shown with moffs indicates its size, which is determined by the address-size attribute of the instruction.
- **Sreg** — A segment register. The segment register bit assignments are ES = 0, CS = 1, SS = 2, DS = 3, FS = 4, and GS = 5.
- **m32fp, m64fp, m80fp** — A single precision, double precision, and double extended-precision (respectively) floating-point operand in memory. These symbols designate floating-point values that are used as operands for x87 FPU floating-point instructions.
- **m16int, m32int, m64int** — A word, doubleword, and quadword integer (respectively) operand in memory. These symbols designate integers that are used as operands for x87 FPU integer instructions.
- **ST or ST(0)** — The top element of the FPU register stack.
- **ST(i)** — The i^{th} element from the top of the FPU register stack (*i* := 0 through 7).
- **mm** — An MMX register. The 64-bit MMX registers are: MM0 through MM7.

- **mm/m32** — The low order 32 bits of an MMX register or a 32-bit memory operand. The 64-bit MMX registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.
- **mm/m64** — An MMX register or a 64-bit memory operand. The 64-bit MMX registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.
- **xmm** — An XMM register. The 128-bit XMM registers are: XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode.
- **xmm/m32**— An XMM register or a 32-bit memory operand. The 128-bit XMM registers are XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.
- **xmm/m64** — An XMM register or a 64-bit memory operand. The 128-bit SIMD floating-point registers are XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.
- **xmm/m128** — An XMM register or a 128-bit memory operand. The 128-bit XMM registers are XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.
- **<XMM0>**— Indicates implied use of the XMM0 register.

  When there is ambiguity, xmm1 indicates the first source operand using an XMM register and xmm2 the second source operand using an XMM register.

  Some instructions use the XMM0 register as the third source operand, indicated by <XMM0>. The use of the third XMM register operand is implicit in the instruction encoding and does not affect the ModR/M encoding.
- **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 instructions.
- **ymm/m256** — A YMM register or 256-bit memory operand.
- **<YMM0>**— Indicates use of the YMM0 register as an implicit argument.
- **bnd** — A 128-bit bounds register. BND0 through BND3.
- **mib** — A memory operand using SIB addressing form, where the index register is not used in address calculation, Scale is ignored. Only the base and displacement are used in effective address calculation.
- **m512** — A 64-byte operand in memory.
- **zmm/m512** — A ZMM register or 512-bit memory operand.
- **{k1}{z}** — A mask register used as instruction writemask. The 64-bit k registers are: k1 through k7. Writemask specification is available exclusively via EVEX prefix. The masking can either be done as a merging-masking, where the old values are preserved for masked out elements or as a zeroing masking. The type of masking is determined by using the EVEX.z bit.
- **{k1}** — Without {z}: a mask register used as instruction writemask for instructions that do not allow zeroing-masking but support merging-masking. This corresponds to instructions that require the value of the aaa field to be different than 0 (e.g., gather) and store-type instructions which allow only merging-masking.
- **k1** — A mask register used as a regular operand (either destination or source). The 64-bit k registers are: k0 through k7.
- **mV** — A vector memory operand; the operand size is dependent on the instruction.
- **vm32{x,y, z}** — A vector array of memory operands specified using VSIB memory addressing. The array of memory addresses are specified using a common base register, a constant scale factor, and a vector index register with individual elements of 32-bit index value in an XMM register (vm32x), a YMM register (vm32y) or a ZMM register (vm32z).
- **vm64{x,y, z}** — A vector array of memory operands specified using VSIB memory addressing. The array of memory addresses are specified using a common base register, a constant scale factor, and a vector index register with individual elements of 64-bit index value in an XMM register (vm64x), a YMM register (vm64y) or a ZMM register (vm64z).

- **zmm/m512/m32bcst** — An operand that can be a ZMM register, a 512-bit memory location or a 512-bit vector loaded from a 32-bit memory location.
- **zmm/m512/m64bcst** — An operand that can be a ZMM register, a 512-bit memory location or a 512-bit vector loaded from a 64-bit memory location.
- **<ZMM0>** — Indicates use of the ZMM0 register as an implicit argument.
- **{er}** — Indicates support for embedded rounding control, which is only applicable to the register-register form of the instruction. This also implies support for SAE (Suppress All Exceptions).
- **{sae}** — Indicates support for SAE (Suppress All Exceptions). This is used for instructions that support SAE, but do not support embedded rounding control.
- **SRC1** — Denotes the first source operand in the instruction syntax of an instruction encoded with the VEX/EVEX prefix and having two or more source operands.
- **SRC2** — Denotes the second source operand in the instruction syntax of an instruction encoded with the VEX/EVEX prefix and having two or more source operands.
- **SRC3** — Denotes the third source operand in the instruction syntax of an instruction encoded with the VEX/EVEX prefix and having three source operands.
- **SRC** — The source in a single-source instruction.
- **DST** — The destination in an instruction. This field is encoded by reg_field.

In the instruction encoding, the MODRM byte is represented several ways depending on the role it plays. The MODRM byte has 3 fields: 2-bit MODRM.MOD field, a 3-bit MODRM.REG field and a 3-bit MODRM.RM field. When all bits of the MODRM byte have fixed values for an instruction, the 2-hex nibble value of that byte is presented after the opcode in the encoding boxes on the instruction description pages. When only some fields of the MODRM byte must contain fixed values, those values are specified as follows:

- If only the MODRM.MOD must be 0b11, and MODRM.REG and MODRM.RM fields are unrestricted, this is denoted as **11:rrr:bbb**. The **rrr** correspond to the 3-bits of the MODRM.REG field and the **bbb** correspond to the 3-bits of the MODMR.RM field.
- If the MODRM.MOD field is constrained to be a value other than 0b11, i.e., it must be one of 0b00, 0b01, or 0b10, then we use the notation !(11).
- If the MODRM.REG field had a specific required value, e.g., 0b101, that would be denoted as mm:101:bbb.

3.1.1.4 Operand Encoding Column in the Instruction Summary Table

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

EVEX encoded instructions employ compressed disp8*N encoding of the displacement bytes, where N is defined in Table 2-36 and Table 2-37, according to tupletypes. The tupletype for an instruction is listed in the operand encoding definition table where applicable.

NOTES

- The letters in the Op/En column of an instruction apply ONLY to the encoding definition table immediately following the instruction summary table.
- In the encoding definition table, the letter 'r' within a pair of parenthesis denotes the content of the operand will be read by the processor. The letter 'w' within a pair of parenthesis denotes the content of the operand will be updated by the processor.

3.1.1.5 64/32-bit Mode Column in the Instruction Summary Table

The "64/32-bit Mode" column indicates whether the 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.

3.1.1.6 CPUID Support Column in the Instruction Summary Table

The fourth column holds abbreviated CPUID feature flags (e.g., appropriate bit in CPUID.01H:ECX, CPUID.01H:EDX for SSE/SSE2/SSE3/SSSE3/SSE4.1/SSE4.2/AESNI/PCLMULQDQ/AVX/RDRAND support) that indicate processor support for the instruction. If the corresponding flag is '0', the instruction will #UD.

3.1.1.7 Description Column in the Instruction Summary Table

The "Description" column briefly explains forms of the instruction.

3.1.1.8 Description Section

Each instruction is then described by number of information sections. The "Description" section describes the purpose of the instructions and required operands in more detail.

Summary of terms that may be used in the description section:

- **Legacy SSE** — Refers to SSE, SSE2, SSE3, SSSE3, SSE4, AESNI, PCLMULQDQ, and any future instruction sets referencing XMM registers and encoded without a VEX prefix.
- **VEX.vvvv** — The VEX bit field specifying a source or destination register (in 1's complement form).
- **rm_field** — shorthand for the ModR/M *r/m* field and any REX.B.
- **reg_field** — shorthand for the ModR/M *reg* field and any REX.R.

3.1.1.9 Operation Section

The "Operation" section contains an algorithm description (frequently written in pseudo-code) for the instruction. Algorithms are composed of the following elements:

- Comments are enclosed within the symbol pairs "(*" and "*)".
- Compound statements are enclosed in keywords, such as: IF, THEN, ELSE, and FI for an if statement; DO and OD for a do statement; or CASE... OF for a case statement.
- A register name implies the contents of the register. A register name enclosed in brackets implies the contents of the location whose address is contained in that register. For example, ES:[DI] indicates the contents of the location whose ES segment relative address is in register DI. [SI] indicates the contents of the address contained in register SI relative to the SI register's default segment (DS) or the overridden segment.
- Parentheses around the "E" in a general-purpose register name, such as (E)SI, indicates that the offset is read from the SI register if the address-size attribute is 16, from the ESI register if the address-size attribute is 32.

Parentheses around the "R" in a general-purpose register name, (R)SI, in the presence of a 64-bit register definition such as (R)SI, indicates that the offset is read from the 64-bit RSI register if the address-size attribute is 64.

- Brackets are used for memory operands where they mean that the contents of the memory location is a segment-relative offset. For example, [SRC] indicates that the content of the source operand is a segment-relative offset.
- A := B indicates that the value of B is assigned to A.
- The symbols =, ≠, >, <, ≥, and ≤ are relational operators used to compare two values: meaning equal, not equal, greater or equal, less or equal, respectively. A relational expression such as A = B is TRUE if the value of A is equal to B; otherwise it is FALSE.
- The expression "« COUNT" and "» COUNT" indicates that the destination operand should be shifted left or right by the number of bits indicated by the count operand.

The following identifiers are used in the algorithmic descriptions:

- **OperandSize and AddressSize** — The OperandSize identifier represents the operand-size attribute of the instruction, which is 16, 32 or 64-bits. The AddressSize identifier represents the address-size attribute, which is 16, 32 or 64-bits. For example, the following pseudo-code indicates that the operand-size attribute depends on the form of the MOV instruction used.

```
IF Instruction = MOVW
    THEN OperandSize := 16;
ELSE
    IF Instruction = MOVD
        THEN OperandSize := 32;
    ELSE
        IF Instruction = MOVQ
            THEN OperandSize := 64;
        FI;
    FI;
FI;
```

  See "Operand-Size and Address-Size Attributes" in Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for guidelines on how these attributes are determined.
- **StackAddrSize** — Represents the stack address-size attribute associated with the instruction, which has a value of 16, 32 or 64-bits. See "Address-Size Attribute for Stack" in Chapter 6, "Procedure Calls, Interrupts, and Exceptions," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.
- **SRC** — Represents the source operand.
- **DEST** — Represents the destination operand.
- **MAXVL** — The maximum vector register width pertaining to the instruction. This is not the vector-length encoding in the instruction's encoding but is instead determined by the current value of XCR0. For details, refer to the table below. Note that the value of MAXVL is the largest of the features enabled. Future processors may define new bits in XCR0 whose setting may imply other values for MAXVL.

MAXVL Definition

XCR0 Component	MAXVL
XCR0.SSE	128
XCR0.AVX	256
XCR0.{ZMM_Hi256, Hi16_ZMM, OPMASK}	512

The following functions are used in the algorithmic descriptions:

- **ZeroExtend(value)** — Returns a value zero-extended to the operand-size attribute of the instruction. For example, if the operand-size attribute is 32, zero extending a byte value of –10 converts the byte from F6H to

a doubleword value of 000000F6H. If the value passed to the ZeroExtend function and the operand-size attribute are the same size, ZeroExtend returns the value unaltered.

- **SignExtend(value)** — Returns a value sign-extended to the operand-size attribute of the instruction. For example, if the operand-size attribute is 32, sign extending a byte containing the value –10 converts the byte from F6H to a doubleword value of FFFFFFF6H. If the value passed to the SignExtend function and the operand-size attribute are the same size, SignExtend returns the value unaltered.
- **SaturateSignedWordToSignedByte** — Converts a signed 16-bit value to a signed 8-bit value. If the signed 16-bit value is less than –128, it is represented by the saturated value -128 (80H); if it is greater than 127, it is represented by the saturated value 127 (7FH).
- **SaturateSignedDwordToSignedWord** — Converts a signed 32-bit value to a signed 16-bit value. If the signed 32-bit value is less than –32768, it is represented by the saturated value –32768 (8000H); if it is greater than 32767, it is represented by the saturated value 32767 (7FFFH).
- **SaturateSignedWordToUnsignedByte** — Converts a signed 16-bit value to an unsigned 8-bit value. If the signed 16-bit value is less than zero, it is represented by the saturated value zero (00H); if it is greater than 255, it is represented by the saturated value 255 (FFH).
- **SaturateToSignedByte** — Represents the result of an operation as a signed 8-bit value. If the result is less than –128, it is represented by the saturated value –128 (80H); if it is greater than 127, it is represented by the saturated value 127 (7FH).
- **SaturateToSignedWord** — Represents the result of an operation as a signed 16-bit value. If the result is less than –32768, it is represented by the saturated value –32768 (8000H); if it is greater than 32767, it is represented by the saturated value 32767 (7FFFH).
- **SaturateToUnsignedByte** — Represents the result of an operation as a signed 8-bit value. If the result is less than zero it is represented by the saturated value zero (00H); if it is greater than 255, it is represented by the saturated value 255 (FFH).
- **SaturateToUnsignedWord** — Represents the result of an operation as a signed 16-bit value. If the result is less than zero it is represented by the saturated value zero (00H); if it is greater than 65535, it is represented by the saturated value 65535 (FFFFH).
- **LowOrderWord(DEST * SRC)** — Multiplies a word operand by a word operand and stores the least significant word of the doubleword result in the destination operand.
- **HighOrderWord(DEST * SRC)** — Multiplies a word operand by a word operand and stores the most significant word of the doubleword result in the destination operand.
- **Push(value)** — Pushes a value onto the stack. The number of bytes pushed is determined by the operand-size attribute of the instruction. See the "Operation" subsection of the "PUSH—Push Word, Doubleword, or Quadword Onto the Stack" section in Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B.
- **Pop()** — removes the value from the top of the stack and returns it. The statement EAX := Pop(); assigns to EAX the 32-bit value from the top of the stack. Pop will return either a word, a doubleword or a quadword depending on the operand-size attribute. See the "Operation" subsection in the "POP—Pop a Value From the Stack" section of Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B.
- **PopRegisterStack** — Marks the FPU ST(0) register as empty and increments the FPU register stack pointer (TOP) by 1.
- **Switch-Tasks** — Performs a task switch.
- **Bit(BitBase, BitOffset)** — Returns the value of a bit within a bit string. The bit string is a sequence of bits in memory or a register. Bits are numbered from low-order to high-order within registers and within memory bytes. If the BitBase is a register, the BitOffset can be in the range 0 to [15, 31, 63] depending on the mode and register size. See Figure 3-1: the function Bit[RAX, 21] is illustrated.

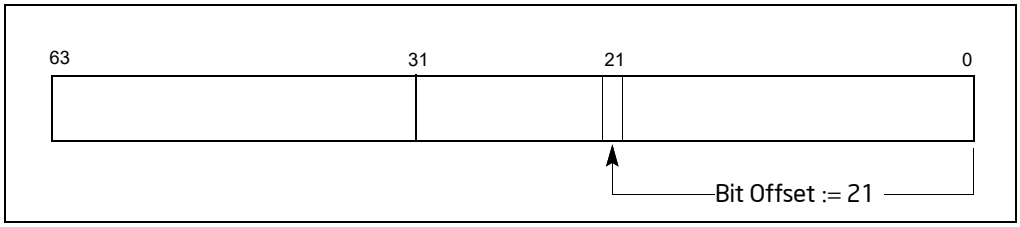


Figure 3-1. Bit Offset for BIT[RAX, 21]

If BitBase is a memory address, the BitOffset has different ranges depending on the operand size (see Table 3-2).

Table 3-2. Range of Bit Positions Specified by Bit Offset Operands

Operand Size	Immediate BitOffset	Register BitOffset
16	0 to 15	-2^{15} to $2^{15}-1$
32	0 to 31	-2^{31} to $2^{31}-1$
64	0 to 63	-2^{63} to $2^{63}-1$

The addressed bit is numbered (Offset MOD 8) within the byte at address (BitBase + (BitOffset DIV 8)) where DIV is signed division with rounding towards negative infinity and MOD returns a positive number (see Figure 3-2).

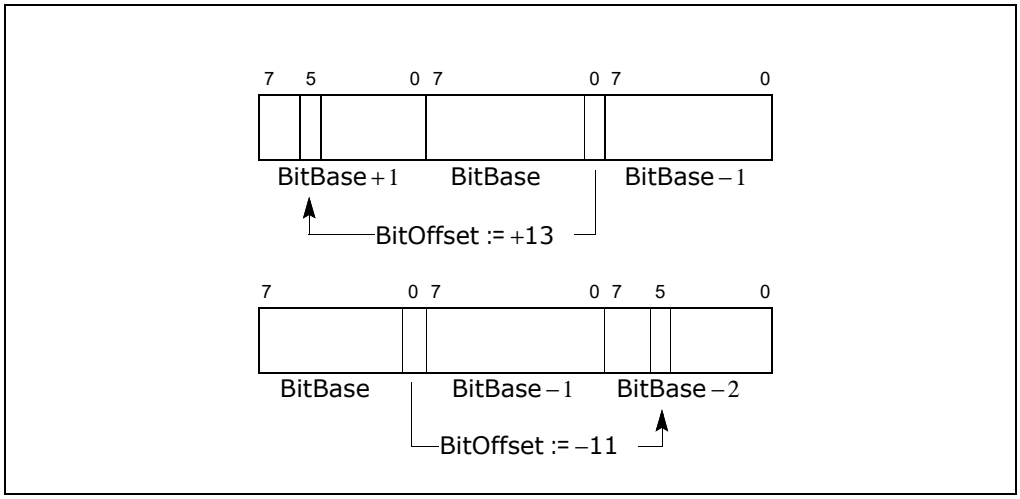


Figure 3-2. Memory Bit Indexing

3.1.1.10 Intel® C/C++ Compiler Intrinsics Equivalents Section

The Intel C/C++ compiler intrinsic functions give access to the full power of the Intel Architecture Instruction Set, while allowing the compiler to optimize register allocation and instruction scheduling for faster execution. Most of these functions are associated with a single IA instruction, although some may generate multiple instructions or different instructions depending upon how they are used. In particular, these functions are used to invoke instructions that perform operations on vector registers that can hold multiple data elements. These SIMD instructions use the following data types.

- __m128, __m256, and __m512 can represent 4, 8, or 16 packed single precision floating-point values, and are used with the vector registers and SSE, AVX, or AVX-512 instruction set extension families. The __m128 data type is also used with various single precision floating-point scalar instructions that perform calculations using

only the lowest 32 bits of a vector register; the remaining bits of the result come from one of the sources or are set to zero depending upon the instruction.

- __m128d, __m256d, and __m512d can represent 2, 4, or 8 packed double precision floating-point values, and are used with the vector registers and SSE, AVX, or AVX-512 instruction set extension families. The __m128d data type is also used with various double precision floating-point scalar instructions that perform calculations using only the lowest 64 bits of a vector register; the remaining bits of the result come from one of the sources or are set to zero depending upon the instruction.
- __m128i, __m256i, and __m512i can represent integer data in bytes, words, doublewords, quadwords, and occasionally larger data types.

Each of these data types incorporates in its name the number of bits it can hold. For example, the __m128 type holds 128 bits, and because each single precision floating-point value is 32 bits long the __m128 type holds (128/32) or four values. Normally the compiler will allocate memory for these data types on an even multiple of the size of the type. Such aligned memory locations may be faster to read and write than locations at other addresses.

These SIMD data types are not basic Standard C data types or C++ objects, so they may be used only with the assignment operator, passed as function arguments, and returned from a function call. If you access the internal members of these types directly, or indirectly by using them in a union, there may be side effects affecting optimization, so it is recommended to use them only with the SIMD instruction intrinsic functions described in this manual or the Intel C/C++ compiler documentation.

Many intrinsic functions names are prefixed with an indicator of the vector length and suffixed by an indicator of the vector element data type, although some functions do not follow the rules below. The prefixes are:

- _mm_ indicates that the function operates on 128-bit (or sometimes 64-bit) vectors.
- _mm256_ indicates the function operates on 256-bit vectors.
- _mm512_ indicates that the function operates on 512-bit vectors.

The suffixes include:

- _ps, which indicates a function that operates on packed single precision floating-point data. Packed single precision floating-point data corresponds to arrays of the C/C++ type *float* with either 4, 8 or 16 elements. Values of this type can be loaded from an array using the *_mm_loadu_ps, _mm256_loadu_ps,* or *_mm512_loadu_ps* functions, or created from individual values using *_mm_set_ps, _mm256_set_ps,* or *_mm512_set_ps* functions, and they can be stored in an array using *_mm_storeu_ps, _mm256_storeu_ps,* or *_mm512_storeu_ps*.
- _ss, which indicates a function that operates on scalar single precision floating-point data. Single precision floating-point data corresponds to the C/C++ type *float*, and values of type float can be converted to type __m128 for use with these functions using the *_mm_set_ss* function, and converted back using the *_mm_cvtss_f32* function. When used with functions that operate on packed single precision floating-point data the scalar element corresponds with the first packed value.
- _pd, which indicates a function that operates on packed double precision floating-point data. Packed double precision floating-point data corresponds to arrays of the C/C++ type *double* with either 2, 4, or 8 elements. Values of this type can be loaded from an array using the *_mm_loadu_pd, _mm256_loadu_pd,* or *_mm512_loadu_pd* functions, or created from individual values using *_mm_set_pd, _mm2566_set_pd,* or *_mm512_set_pd* functions, and they can be stored in an array using *_mm_storeu_pd, _mm256_storeu_pd,* or *_mm512_storeu_pd*.
- _sd, which indicates a function that operates on scalar double precision floating-point data. Double precision floating-point data corresponds to the C/C++ type *double*, and values of type double can be converted to type __m128d for use with these functions using the *_mm_set_sd* function, and converted back using the *_mm_cvtsd_f64* function. When used with functions that operate on packed double precision floating-point data the scalar element corresponds with the first packed value.
- _epi8, which indicates a function that operates on packed 8-bit signed integer values. Packed 8-bit signed integers correspond to an array of *signed char* with 16, 32 or 64 elements. Values of this type can be created from individual elements using *_mm_set_epi8, _mm256_set_epi8,* or *_mm512_set_epi8* functions.
- _epi16, which indicates a function that operates on packed 16-bit signed integer values. Packed 16-bit signed integers correspond to an array of *short* with 8, 16 or 32 elements. Values of this type can be created from individual elements using *_mm_set_epi16, _mm256_set_epi16,* or *_mm512_set_epi16* functions.

- _epi32, which indicates a function that operates on packed 32-bit signed integer values. Packed 32-bit signed integers correspond to an array of *int* with 4, 8 or 16 elements. Values of this type can be created from individual elements using *_mm_set_epi32, _mm256_set_epi32,* or *_mm512_set_epi32* functions.
- _epi64, which indicates a function that operates on packed 64-bit signed integer values. Packed 64-bit signed integers correspond to an array of *long long* (or *long* if it is a 64-bit data type) with 2, 4 or 8 elements. Values of this type can be created from individual elements using *_mm_set_epi32, _mm256_set_epi32,* or *_mm512_set_epi32* functions.
- _epu8, which indicates a function that operates on packed 8-bit unsigned integer values. Packed 8-bit unsigned integers correspond to an array of *unsigned char* with 16, 32 or 64 elements.
- _epu16, which indicates a function that operates on packed 16-bit unsigned integer values. Packed 16-bit unsigned integers correspond to an array of *unsigned short* with 8, 16 or 32 elements.
- _epu32, which indicates a function that operates on packed 32-bit unsigned integer values. Packed 32-bit unsigned integers correspond to an array of *unsigned* with 4, 8 or 16 elements.
- _epu64, which indicates a function that operates on packed 64-bit unsigned integer values. Packed 64-bit unsigned integers correspond to an array of *unsigned long long* (or *unsigned long* if it is a 64-bit data type) with 2, 4 or 8 elements.
- _si128, which indicates a function that operates on a single 128-bit value of type __m128i.
- _si256, which indicates a function that operates on a single a 256-bit value of type __m256i.
- _si512, which indicates a function that operates on a single a 512-bit value of type __m512i.

Values of any packed integer type can be loaded from an array using the *_mm_loadu_si128, _mm256_loadu_-si256,* or *_mm512_loadu_si512* functions, and they can be stored in an array using *_mm_storeu_si128, _mm256_storeu_si256,* or *_mm512_storeu_si512*.

These functions and data types are used with the SSE, AVX, and AVX-512 instruction set extension families. In addition there are similar functions that correspond to MMX instructions. These are less frequently used because they require additional state management, and only operate on 64-bit packed integer values.

The declarations of Intel C/C++ compiler intrinsic functions may reference some non-standard data types, such as __int64. The C Standard header *stdint.h* defines similar platform-independent types, and the documentation for that header gives characteristics that apply to corresponding non-standard types according to the following table.

Table 3-3. Standard and Non-Standard Data Types

Non-standard Type	Standard Type (from stdint.h)
__int64	int64_t
unsigned __int64	uint64_t
__int32	int32_t
unsigned __int32	uint32_t
__int16	int16_t
unsigned __int16	uint16_t

For a more detailed description of each intrinsic function and additional information related to its usage, refer to the online Intel Intrinsics Guide, https://software.intel.com/sites/landingpage/IntrinsicsGuide.

3.1.1.11 Flags Affected Section

The "Flags Affected" section lists the flags in the EFLAGS register that are affected by the instruction. When a flag is cleared, it is equal to 0; when it is set, it is equal to 1. The arithmetic and logical instructions usually assign values to the status flags in a uniform manner (see Appendix A, "EFLAGS Cross-Reference," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1). Non-conventional assignments are described in the "Operation" section. The values of flags listed as **undefined** may be changed by the instruction in an indeterminate manner. Flags that are not listed are unchanged by the instruction.

3.1.1.12 FPU Flags Affected Section

The floating-point instructions have an "FPU Flags Affected" section that describes how each instruction can affect the four condition code flags of the FPU status word.

3.1.1.13 Protected Mode Exceptions Section

The "Protected Mode Exceptions" section lists the exceptions that can occur when the instruction is executed in protected mode and the reasons for the exceptions. Each exception is given a mnemonic that consists of a pound sign (#) followed by two letters and an optional error code in parentheses. For example, #GP(0) denotes a general protection exception with an error code of 0. Table 3-4 associates each two-letter mnemonic with the corresponding exception vector and name. See Chapter 6, "Procedure Calls, Interrupts, and Exceptions," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, for a detailed description of the exceptions.

Application programmers should consult the documentation provided with their operating systems to determine the actions taken when exceptions occur.

Table 3-4. Intel 64® and IA-32 General Exceptions

Vector	Name	Source	Protected Mode[1]	Real Address Mode	Virtual 8086 Mode
0	#DE—Divide Error	DIV and IDIV instructions.	Yes	Yes	Yes
1	#DB—Debug	Any code or data reference.	Yes	Yes	Yes
3	#BP—Breakpoint	INT3 instruction.	Yes	Yes	Yes
4	#OF—Overflow	INTO instruction.	Yes	Yes	Yes
5	#BR—BOUND Range Exceeded	BOUND instruction.	Yes	Yes	Yes
6	#UD—Invalid Opcode (Undefined Opcode)	UD instruction or reserved opcode.	Yes	Yes	Yes
7	#NM—Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.	Yes	Yes	Yes
8	#DF—Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.	Yes	Yes	Yes
10	#TS—Invalid TSS	Task switch or TSS access.	Yes	No	Yes
11	#NP—Segment Not Present	Loading segment registers or accessing system segments.	Yes	No	Yes
12	#SS—Stack Segment Fault	Stack operations and SS register loads.	Yes	Yes	Yes
13	#GP—General Protection[2]	Any memory reference and other protection checks.	Yes	Yes	Yes
14	#PF—Page Fault	Any memory reference.	Yes	No	Yes
16	#MF—Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.	Yes	Yes	Yes
17	#AC—Alignment Check	Any data reference in memory.	Yes	No	Yes
18	#MC—Machine Check	Model dependent machine check errors.	Yes	Yes	Yes
19	#XM—SIMD Floating-Point Numeric Error	SSE/SSE2/SSE3 floating-point instructions.	Yes	Yes	Yes
20	#VE—Virtualization Exception	EPT violations[3]	Yes	No	No

Table 3-4. Intel 64® and IA-32 General Exceptions (Contd.)

Vector	Name	Source	Protected Mode[1]	Real Address Mode	Virtual 8086 Mode
21	#CP—Control Protection Exception	RET, IRET, RSTORSSP, and SETSSBSY instructions can generate this exception. When CET indirect branch tracking is enabled, this exception can be generated due to a missing ENDBRANCH instruction at target of an indirect call or jump.	Yes	No	No

NOTES:

1. Apply to protected mode, compatibility mode, and 64-bit mode.
2. In the real-address mode, vector 13 is the segment overrun exception.
3. This exception can occur only on processors that support the 1-setting of the "EPT-violation #VE" VM-execution control.

3.1.1.14 Real-Address Mode Exceptions Section

The "Real-Address Mode Exceptions" section lists the exceptions that can occur when the instruction is executed in real-address mode (see Table 3-4).

3.1.1.15 Virtual-8086 Mode Exceptions Section

The "Virtual-8086 Mode Exceptions" section lists the exceptions that can occur when the instruction is executed in virtual-8086 mode (see Table 3-4).

3.1.1.16 Floating-Point Exceptions Section

The "Floating-Point Exceptions" section lists exceptions that can occur when an x87 FPU floating-point instruction is executed. All of these exception conditions result in a floating-point error exception (#MF, exception 16) being generated. Table 3-5 associates a one- or two-letter mnemonic with the corresponding exception name. See "Floating-Point Exception Conditions" in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for a detailed description of these exceptions.

Table 3-5. x87 FPU Floating-Point Exceptions

Mnemonic	Name	Source
#IS #IA	Floating-point invalid operation: - Stack overflow or underflow - Invalid arithmetic operation	 - x87 FPU stack overflow or underflow - Invalid FPU arithmetic operation
#Z	Floating-point divide-by-zero	Divide-by-zero
#D	Floating-point denormal operand	Source operand that is a denormal number
#O	Floating-point numeric overflow	Overflow in result
#U	Floating-point numeric underflow	Underflow in result
#P	Floating-point inexact result (precision)	Inexact result (precision)

3.1.1.17 SIMD Floating-Point Exceptions Section

The "SIMD Floating-Point Exceptions" section lists exceptions that can occur when an SSE/SSE2/SSE3 floating-point instruction is executed. All of these exception conditions result in a SIMD floating-point error exception (#XM, exception 19) being generated. Table 3-6 associates a one-letter mnemonic with the corresponding exception name. For a detailed description of these exceptions, refer to "SSE and SSE2 Exceptions", in Chapter 11 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Table 3-6. SIMD Floating-Point Exceptions

Mnemonic	Name	Source
#I	Floating-point invalid operation	Invalid arithmetic operation or source operand
#Z	Floating-point divide-by-zero	Divide-by-zero
#D	Floating-point denormal operand	Source operand that is a denormal number
#O	Floating-point numeric overflow	Overflow in result
#U	Floating-point numeric underflow	Underflow in result
#P	Floating-point inexact result	Inexact result (precision)

3.1.1.18 Compatibility Mode Exceptions Section

This section lists exceptions that occur within compatibility mode.

3.1.1.19 64-Bit Mode Exceptions Section

This section lists exceptions that occur within 64-bit mode.

3.2 INTEL® AMX CONSIDERATIONS

The following implementation parameters and helper functions are applicable to the Intel® AMX instructions.

3.2.1 Implementation Parameters

The parameters are reported via CPUID.1DH. Index 0 reports all zeros for all fields.

```
define palette_table[id]:
    uint16_t total_tile_bytes
    uint16_t bytes_per_tile
    uint16_t bytes_per_row
    uint16_t max_names
    uint16_t max_rows
```

The tile parameters are set by LDTILECFG or XRSTOR* of TILECFG:

```
define tile[tid]:
    byte rows
    word colsb // bytes_per_row
    bool valid
```

3.2.2 Helper Functions

The helper functions used in Intel AMX instructions are defined below.

```
define write_row_and_zero(treg, r, data, nbytes):
   for j in 0 ...nbytes-1:
       treg.row[r].byte[j] := data.byte[j]

   // zero the rest of the row
   for j in nbytes ... palette_table[tilecfg.palette_id].bytes_per_row-1:
      treg.row[r].byte[j] := 0


define zero_upper_rows(treg, r):
   for i in r ... palette_table[tilecfg.palette_id].max_rows-1:
      for j in 0 ... palette_table[tilecfg.palette_id].bytes_per_row-1:
                  treg.row[i].byte[j] := 0



define zero_tilecfg_start():
   tilecfg.start_row :=0


define zero_all_tile_data():
  if XCR0[TILEDATA]:
      b := CPUID.0DH.TILEDATA:EAX // size of feature
      for j in 0 ...b:
             TILEDATA.byte[j] := 0

define xcr0_supports_palette(palette_id):
  if palette_id == 0:
     return 1
  elif palette_id == 1:
     if XCR0[TILECFG] and XCR0[TILEDATA]:
        return 1
  return 0
```

3.3 INSTRUCTIONS (A-L)

The remainder of this chapter provides descriptions of Intel 64 and IA-32 instructions (A-L). See also: Chapter 4, "Instruction Set Reference, M-U," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B; Chapter 5, "Instruction Set Reference, V," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C; and Chapter 6, "Instruction Set Reference, W-Z," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D.

AAA—ASCII Adjust After Addition

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
37	AAA	ZO	Invalid	Valid	ASCII adjust AL after addition.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Adjusts the sum of two unpacked BCD values to create an unpacked BCD result. The AL register is the implied source and destination operand for this instruction. The AAA instruction is only useful when it follows an ADD instruction that adds (binary addition) two unpacked BCD values and stores a byte result in the AL register. The AAA instruction then adjusts the contents of the AL register to contain the correct 1-digit unpacked BCD result.

If the addition produces a decimal carry, the AH register increments by 1, and the CF and AF flags are set. If there was no decimal carry, the CF and AF flags are cleared and the AH register is unchanged. In either case, bits 4 through 7 of the AL register are set to 0.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

```
IF 64-Bit Mode
    THEN
        #UD;
    ELSE
        IF ((AL AND 0FH) > 9) or (AF = 1)
            THEN
                AX := AX + 106H;
                AF := 1;
                CF := 1;
            ELSE
                AF := 0;
                CF := 0;
        FI;
        AL := AL AND 0FH;
FI;
```

Flags Affected

The AF and CF flags are set to 1 if the adjustment results in a decimal carry; otherwise they are set to 0. The OF, SF, ZF, and PF flags are undefined.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as protected mode.

Compatibility Mode Exceptions

Same exceptions as protected mode.

64-Bit Mode Exceptions

#UD	If in 64-bit mode.

AAD—ASCII Adjust AX Before Division

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
D5 0A	AAD	ZO	Invalid	Valid	ASCII adjust AX before division.
D5 ib	AAD imm8	ZO	Invalid	Valid	Adjust AX before division to number base imm8.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Adjusts two unpacked BCD digits (the least-significant digit in the AL register and the most-significant digit in the AH register) so that a division operation performed on the result will yield a correct unpacked BCD value. The AAD instruction is only useful when it precedes a DIV instruction that divides (binary division) the adjusted value in the AX register by an unpacked BCD value.

The AAD instruction sets the value in the AL register to (AL + (10 * AH)), and then clears the AH register to 00H. The value in the AX register is then equal to the binary equivalent of the original unpacked two-digit (base 10) number in registers AH and AL.

The generalized version of this instruction allows adjustment of two unpacked digits of any number base (see the "Operation" section below), by setting the *imm8* byte to the selected number base (for example, 08H for octal, 0AH for decimal, or 0CH for base 12 numbers). The AAD mnemonic is interpreted by all assemblers to mean adjust ASCII (base 10) values. To adjust values in another number base, the instruction must be hand coded in machine code (D5 *imm8*).

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

```
IF 64-Bit Mode
    THEN
        #UD;
    ELSE
        tempAL := AL;
        tempAH := AH;
        AL := (tempAL + (tempAH * imm8)) AND FFH;
        (* imm8 is set to 0AH for the AAD mnemonic.*)
        AH := 0;
FI;
```

The immediate value (*imm8*) is taken from the second byte of the instruction.

Flags Affected

The SF, ZF, and PF flags are set according to the resulting binary value in the AL register; the OF, AF, and CF flags are undefined.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as protected mode.

Compatibility Mode Exceptions

Same exceptions as protected mode.

64-Bit Mode Exceptions

#UD	If in 64-bit mode.

AAM—ASCII Adjust AX After Multiply

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
D4 0A	AAM	ZO	Invalid	Valid	ASCII adjust AX after multiply.
D4 ib	AAM imm8	ZO	Invalid	Valid	Adjust AX after multiply to number base imm8.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Adjusts the result of the multiplication of two unpacked BCD values to create a pair of unpacked (base 10) BCD values. The AX register is the implied source and destination operand for this instruction. The AAM instruction is only useful when it follows an MUL instruction that multiplies (binary multiplication) two unpacked BCD values and stores a word result in the AX register. The AAM instruction then adjusts the contents of the AX register to contain the correct 2-digit unpacked (base 10) BCD result.

The generalized version of this instruction allows adjustment of the contents of the AX to create two unpacked digits of any number base (see the "Operation" section below). Here, the *imm8* byte is set to the selected number base (for example, 08H for octal, 0AH for decimal, or 0CH for base 12 numbers). The AAM mnemonic is interpreted by all assemblers to mean adjust to ASCII (base 10) values. To adjust to values in another number base, the instruction must be hand coded in machine code (D4 *imm8*).

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

```
IF 64-Bit Mode
    THEN
        #UD;
    ELSE
        tempAL := AL;
        AH := tempAL / imm8; (* imm8 is set to 0AH for the AAM mnemonic *)
        AL := tempAL MOD imm8;
FI;
```

The immediate value (*imm8*) is taken from the second byte of the instruction.

Flags Affected

The SF, ZF, and PF flags are set according to the resulting binary value in the AL register. The OF, AF, and CF flags are undefined.

Protected Mode Exceptions

#DE	If an immediate value of 0 is used.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as protected mode.

Compatibility Mode Exceptions

Same exceptions as protected mode.

64-Bit Mode Exceptions

#UD	If in 64-bit mode.

AAS—ASCII Adjust AL After Subtraction

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
3F	AAS	ZO	Invalid	Valid	ASCII adjust AL after subtraction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Adjusts the result of the subtraction of two unpacked BCD values to create a unpacked BCD result. The AL register is the implied source and destination operand for this instruction. The AAS instruction is only useful when it follows a SUB instruction that subtracts (binary subtraction) one unpacked BCD value from another and stores a byte result in the AL register. The AAA instruction then adjusts the contents of the AL register to contain the correct 1-digit unpacked BCD result.

If the subtraction produced a decimal carry, the AH register decrements by 1, and the CF and AF flags are set. If no decimal carry occurred, the CF and AF flags are cleared, and the AH register is unchanged. In either case, the AL register is left with its top four bits set to 0.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

```
IF 64-bit mode
    THEN
        #UD;
    ELSE
        IF ((AL AND 0FH) > 9) or (AF = 1)
            THEN
                AX := AX - 6;
                AH := AH - 1;
                AF := 1;
                CF := 1;
                AL := AL AND 0FH;
            ELSE
                CF := 0;
                AF := 0;
                AL := AL AND 0FH;
        FI;
FI;
```

Flags Affected

The AF and CF flags are set to 1 if there is a decimal borrow; otherwise, they are cleared to 0. The OF, SF, ZF, and PF flags are undefined.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as protected mode.

Compatibility Mode Exceptions

Same exceptions as protected mode.

64-Bit Mode Exceptions

#UD	If in 64-bit mode.

ADC—Add With Carry

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
14 ib	ADC AL, imm8	I	Valid	Valid	Add with carry imm8 to AL.
15 iw	ADC AX, imm16	I	Valid	Valid	Add with carry imm16 to AX.
15 id	ADC EAX, imm32	I	Valid	Valid	Add with carry imm32 to EAX.
REX.W + 15 id	ADC RAX, imm32	I	Valid	N.E.	Add with carry imm32 sign extended to 64-bits to RAX.
80 /2 ib	ADC r/m8[1], imm8	MI	Valid	Valid	Add with carry imm8 to r/m8.
81 /2 iw	ADC r/m16, imm16	MI	Valid	Valid	Add with carry imm16 to r/m16.
81 /2 id	ADC r/m32, imm32	MI	Valid	Valid	Add with CF imm32 to r/m32.
REX.W + 81 /2 id	ADC r/m64, imm32	MI	Valid	N.E.	Add with CF imm32 sign extended to 64-bits to r/m64.
83 /2 ib	ADC r/m16, imm8	MI	Valid	Valid	Add with CF sign-extended imm8 to r/m16.
83 /2 ib	ADC r/m32, imm8	MI	Valid	Valid	Add with CF sign-extended imm8 into r/m32.
REX.W + 83 /2 ib	ADC r/m64, imm8	MI	Valid	N.E.	Add with CF sign-extended imm8 into r/m64.
10 /r	ADC r/m8[1], r8[1]	MR	Valid	Valid	Add with carry byte register to r/m8.
11 /r	ADC r/m16, r16	MR	Valid	Valid	Add with carry r16 to r/m16.
11 /r	ADC r/m32, r32	MR	Valid	Valid	Add with CF r32 to r/m32.
REX.W + 11 /r	ADC r/m64, r64	MR	Valid	N.E.	Add with CF r64 to r/m64.
12 /r	ADC r8[1], r/m8[1]	RM	Valid	Valid	Add with carry r/m8 to byte register.
13 /r	ADC r16, r/m16	RM	Valid	Valid	Add with carry r/m16 to r16.
13 /r	ADC r32, r/m32	RM	Valid	Valid	Add with CF r/m32 to r32.
REX.W + 13 /r	ADC r64, r/m64	RM	Valid	N.E.	Add with CF r/m64 to r64.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
MR	ModRM:r/m (r, w)	ModRM:reg (r)	N/A	N/A
MI	ModRM:r/m (r, w)	imm8/16/32	N/A	N/A
I	AL/AX/EAX/RAX	imm8/16/32	N/A	N/A

Description

Adds the destination operand (first operand), the source operand (second operand), and the carry (CF) flag and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) The state of the CF flag represents a carry from a previous addition. When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADC instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a carry in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

The ADC instruction is usually executed as part of a multibyte or multiword addition in which an ADD instruction is followed by an ADC instruction.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

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

Operation

```
DEST := DEST + SRC + CF;
```

Intel C/C++ Compiler Intrinsic Equivalent

ADC extern unsigned char _addcarry_u8(unsigned char c_in, unsigned char src1, unsigned char src2, unsigned char *sum_out);
ADC extern unsigned char _addcarry_u16(unsigned char c_in, unsigned short src1, unsigned short src2, unsigned short *sum_out);
ADC extern unsigned char _addcarry_u32(unsigned char c_in, unsigned int src1, unsigned char int, unsigned int *sum_out);
ADC extern unsigned char _addcarry_u64(unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *sum_out);

Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

#UD	If the LOCK prefix is used but the destination is not a memory operand.

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	RM	V/V	ADX	Unsigned addition of r32 with CF, r/m32 to r32, writes CF.
66 REX.w 0F 38 F6 /r ADCX r64, r/m64	RM	V/N.E.	ADX	Unsigned addition of r64 with CF, r/m64 to r64, writes CF.

Instruction Operand Encoding

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

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 the 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.

Protected Mode Exceptions

#UD	If the LOCK prefix is used. If CPUID.07H.00H:EBX.ADX[19] = 0.
#SS(0)	For an illegal address in the SS segment.
#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used. If CPUID.07H.00H:EBX.AVX[19] = 0.
#SS(0)	For an illegal address in the SS segment.
#GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used. If CPUID.07H.00H:EBX.AVX[19] = 0.
#SS(0)	For an illegal address in the SS segment.
#GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used. If CPUID.07H.00H:EBX.AVX[19] = 0.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

ADD—Add

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
04 ib	ADD AL, imm8	I	Valid	Valid	Add imm8 to AL.
05 iw	ADD AX, imm16	I	Valid	Valid	Add imm16 to AX.
05 id	ADD EAX, imm32	I	Valid	Valid	Add imm32 to EAX.
REX.W + 05 id	ADD RAX, imm32	I	Valid	N.E.	Add imm32 sign-extended to 64-bits to RAX.
80 /0 ib	ADD r/m8[1], imm8	MI	Valid	Valid	Add imm8 to r/m8.
81 /0 iw	ADD r/m16, imm16	MI	Valid	Valid	Add imm16 to r/m16.
81 /0 id	ADD r/m32, imm32	MI	Valid	Valid	Add imm32 to r/m32.
REX.W + 81 /0 id	ADD r/m64, imm32	MI	Valid	N.E.	Add imm32 sign-extended to 64-bits to r/m64.
83 /0 ib	ADD r/m16, imm8	MI	Valid	Valid	Add sign-extended imm8 to r/m16.
83 /0 ib	ADD r/m32, imm8	MI	Valid	Valid	Add sign-extended imm8 to r/m32.
REX.W + 83 /0 ib	ADD r/m64, imm8	MI	Valid	N.E.	Add sign-extended imm8 to r/m64.
00 /r	ADD r/m8[1], r8[1]	MR	Valid	Valid	Add r8 to r/m8.
01 /r	ADD r/m16, r16	MR	Valid	Valid	Add r16 to r/m16.
01 /r	ADD r/m32, r32	MR	Valid	Valid	Add r32 to r/m32.
REX.W + 01 /r	ADD r/m64, r64	MR	Valid	N.E.	Add r64 to r/m64.
02 /r	ADD r8[1], r/m8[1]	RM	Valid	Valid	Add r/m8 to r8.
03 /r	ADD r16, r/m16	RM	Valid	Valid	Add r/m16 to r16.
03 /r	ADD r32, r/m32	RM	Valid	Valid	Add r/m32 to r32.
REX.W + 03 /r	ADD r64, r/m64	RM	Valid	N.E.	Add r/m64 to r64.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
MR	ModRM:r/m (r, w)	ModRM:reg (r)	N/A	N/A
MI	ModRM:r/m (r, w)	imm8/16/32	N/A	N/A
I	AL/AX/EAX/RAX	imm8/16/32	N/A	N/A

Description

Adds the destination operand (first operand) and the source operand (second operand) and then stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADD instruction performs integer addition. It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate a carry (overflow) in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

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

Operation

```
DEST := DEST + SRC;
```

Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

ADDPD—Add Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 58 /r ADDPD xmm1, xmm2/m128	A	V/V	SSE2	Add packed double precision floating-point values from xmm2/mem to xmm1 and store result in xmm1.
VEX.128.66.0F.WIG 58 /r VADDPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Add packed double precision floating-point values from xmm3/mem to xmm2 and store result in xmm1.
VEX.256.66.0F.WIG 58 /r VADDPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Add packed double precision floating-point values from ymm3/mem to ymm2 and store result in ymm1.
EVEX.128.66.0F.W1 58 /r VADDPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Add packed double precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1 with writemask k1.
EVEX.256.66.0F.W1 58 /r VADDPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Add packed double precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1 with writemask k1.
EVEX.512.66.0F.W1 58 /r VADDPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst {er}	C	V/V	AVX512F OR AVX10.1	Add packed double precision floating-point values from zmm3/m512/m64bcst to zmm2 and store result in zmm1 with writemask k1.

Instruction Operand Encoding

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

Description

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

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

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

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

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

Operation

VADDPD (EVEX Encoded Versions) When SRC2 Operand is a Vector Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC1[i+63:i] + SRC2[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VADDPD (EVEX Encoded Versions) When SRC2 Operand is a Memory Source

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

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

VADDPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0] + SRC2[63:0]
DEST[127:64] := SRC1[127:64] + SRC2[127:64]
DEST[191:128] := SRC1[191:128] + SRC2[191:128]
DEST[255:192] := SRC1[255:192] + SRC2[255:192]
DEST[MAXVL-1:256] := 0
```

.

VADDPD (VEX.128 Encoded Version)
DEST[63:0] := SRC1[63:0] + SRC2[63:0]
DEST[127:64] := SRC1[127:64] + SRC2[127:64]
DEST[MAXVL-1:128] := 0

ADDPD (128-bit Legacy SSE Version)
DEST[63:0] := DEST[63:0] + SRC[63:0]
DEST[127:64] := DEST[127:64] + SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VADDPD __m512d _mm512_add_pd (__m512d a, __m512d b);
VADDPD __m512d _mm512_mask_add_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);
VADDPD __m512d _mm512_maskz_add_pd (__mmask8 k, __m512d a, __m512d b);
VADDPD __m256d _mm256_mask_add_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);
VADDPD __m256d _mm256_maskz_add_pd (__mmask8 k, __m256d a, __m256d b);
VADDPD __m128d _mm_mask_add_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VADDPD __m128d _mm_maskz_add_pd (__mmask8 k, __m128d a, __m128d b);
VADDPD __m512d _mm512_add_round_pd (__m512d a, __m512d b, int);
VADDPD __m512d _mm512_mask_add_round_pd (__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VADDPD __m512d _mm512_maskz_add_round_pd (__mmask8 k, __m512d a, __m512d b, int);
ADDPD __m256d _mm256_add_pd (__m256d a, __m256d b);
ADDPD __m128d _mm_add_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instruction, see Table 2-19, "Type 2 Class Exception Conditions."

EVEX-encoded instruction, see Table 2-48, "Type E2 Class Exception Conditions."

ADDPS—Add Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 58 /r ADDPS xmm1, xmm2/m128	A	V/V	SSE	Add packed single precision floating-point values from xmm2/m128 to xmm1 and store result in xmm1.
VEX.128.0F.WIG 58 /r VADDPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Add packed single precision floating-point values from xmm3/m128 to xmm2 and store result in xmm1.
VEX.256.0F.WIG 58 /r VADDPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Add packed single precision floating-point values from ymm3/m256 to ymm2 and store result in ymm1.
EVEX.128.0F.W0 58 /r VADDPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Add packed single precision floating-point values from xmm3/m128/m32bcst to xmm2 and store result in xmm1 with writemask k1.
EVEX.256.0F.W0 58 /r VADDPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Add packed single precision floating-point values from ymm3/m256/m32bcst to ymm2 and store result in ymm1 with writemask k1.
EVEX.512.0F.W0 58 /r VADDPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst {er}	C	V/V	AVX512F OR AVX10.1	Add packed single precision floating-point values from zmm3/m512/m32bcst to zmm2 and store result in zmm1 with writemask k1.

Instruction Operand Encoding

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

Description

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

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

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

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

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

Operation

VADDPS (EVEX Encoded Versions) When SRC2 Operand is a Register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC1[i+31:i] + SRC2[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VADDPS (EVEX Encoded Versions) When SRC2 Operand is a Memory Source

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

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] := SRC1[i+31:i] + SRC2[31:0]
                ELSE
                    DEST[i+31:i] := SRC1[i+31:i] + SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VADDPS (VEX.256 Encoded Version)
DEST[31:0] := SRC1[31:0] + SRC2[31:0]
DEST[63:32] := SRC1[63:32] + SRC2[63:32]
DEST[95:64] := SRC1[95:64] + SRC2[95:64]
DEST[127:96] := SRC1[127:96] + SRC2[127:96]
DEST[159:128] := SRC1[159:128] + SRC2[159:128]
DEST[191:160]:= SRC1[191:160] + SRC2[191:160]
DEST[223:192] := SRC1[223:192] + SRC2[223:192]
DEST[255:224] := SRC1[255:224] + SRC2[255:224].
DEST[MAXVL-1:256] := 0

VADDPS (VEX.128 Encoded Version)
DEST[31:0] := SRC1[31:0] + SRC2[31:0]
DEST[63:32] := SRC1[63:32] + SRC2[63:32]
DEST[95:64] := SRC1[95:64] + SRC2[95:64]
DEST[127:96] := SRC1[127:96] + SRC2[127:96]
DEST[MAXVL-1:128] := 0

ADDPS (128-bit Legacy SSE Version)
DEST[31:0] := DEST[31:0] + SRC[31:0]
DEST[63:32] := DEST[63:32] + SRC[63:32]
DEST[95:64] := DEST[95:64] + SRC[95:64]
DEST[127:96] := DEST[127:96] + SRC[127:96]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VADDPS __m512 _mm512_add_ps (__m512 a, __m512 b);
VADDPS __m512 _mm512_mask_add_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VADDPS __m512 _mm512_maskz_add_ps (__mmask16 k, __m512 a, __m512 b);
VADDPS __m256 _mm256_mask_add_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VADDPS __m256 _mm256_maskz_add_ps (__mmask8 k, __m256 a, __m256 b);
VADDPS __m128 _mm_mask_add_ps (__m128d s, __mmask8 k, __m128 a, __m128 b);
VADDPS __m128 _mm_maskz_add_ps (__mmask8 k, __m128 a, __m128 b);
VADDPS __m512 _mm512_add_round_ps (__m512 a, __m512 b, int);
VADDPS __m512 _mm512_mask_add_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VADDPS __m512 _mm512_maskz_add_round_ps (__mmask16 k, __m512 a, __m512 b, int);
ADDPS __m256 _mm256_add_ps (__m256 a, __m256 b);
ADDPS __m128 _mm_add_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instruction, see Table 2-19, "Type 2 Class Exception Conditions."

EVEX-encoded instruction, see Table 2-48, "Type E2 Class Exception Conditions."

ADDSD—Add Scalar Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	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.LIG.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.
EVEX.LLIG.F2.0F.W1 58 /r VADDSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F OR AVX10.1	Add the low double precision floating-point value from xmm3/m64 to xmm2 and store the result in xmm1 with writemask k1.

Instruction Operand Encoding

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

Description

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

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

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

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

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

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

Operation

VADDSD (EVEX Encoded Version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := SRC1[63:0] + SRC2[63:0]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VADDSD (VEX.128 Encoded Version)

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

ADDSD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] + SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VADDSD __m128d _mm_mask_add_sd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VADDSD __m128d _mm_maskz_add_sd (__mmask8 k, __m128d a, __m128d b);
VADDSD __m128d _mm_add_round_sd (__m128d a, __m128d b, int);
VADDSD __m128d _mm_mask_add_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VADDSD __m128d _mm_maskz_add_round_sd (__mmask8 k, __m128d a, __m128d b, int);
ADDSD __m128d _mm_add_sd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instruction, see Table 2-20, "Type 3 Class Exception Conditions."

EVEX-encoded instruction, see Table 2-49, "Type E3 Class Exception Conditions."

ADDSS—Add Scalar Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 58 /r ADDSS xmm1, xmm2/m32	A	V/V	SSE	Add the low single precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1.
VEX.LIG.F3.0F.WIG 58 /r VADDSS xmm1,xmm2, xmm3/m32	B	V/V	AVX	Add the low single precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1.
EVEX.LLIG.F3.0F.W0 58 /r VADDSS xmm1{k1}{z}, xmm2, xmm3/m32{er}	C	V/V	AVX512F OR AVX10.1	Add the low single precision floating-point value from xmm3/m32 to xmm2 and store the result in xmm1with writemask k1.

Instruction Operand Encoding

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

Description

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

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

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

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

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

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

Operation

VADDSS (EVEX Encoded Versions)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := SRC1[31:0] + SRC2[31:0]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VADDSS DEST, SRC1, SRC2 (VEX.128 Encoded Version)

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

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

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

Intel C/C++ Compiler Intrinsic Equivalent

VADDSS __m128 _mm_mask_add_ss (__m128 s, __mmask8 k, __m128 a, __m128 b);
VADDSS __m128 _mm_maskz_add_ss (__mmask8 k, __m128 a, __m128 b);
VADDSS __m128 _mm_add_round_ss (__m128 a, __m128 b, int);
VADDSS __m128 _mm_mask_add_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VADDSS __m128 _mm_maskz_add_round_ss (__mmask8 k, __m128 a, __m128 b, int);
ADDSS __m128 _mm_add_ss (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instruction, see Table 2-20, "Type 3 Class Exception Conditions."

EVEX-encoded instruction, see Table 2-49, "Type E3 Class Exception Conditions."

ADDSUBPD—Packed Double Precision Floating-Point Add/Subtract

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

Instruction Operand Encoding

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

Description

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

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

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified. See Figure 3-3.

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

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

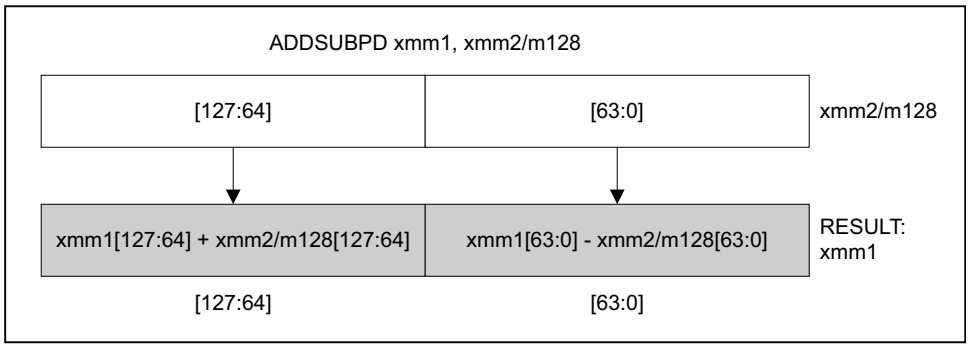


Figure 3-3. ADDSUBPD—Packed Double Precision Floating-Point Add/Subtract

Operation

ADDSUBPD (128-bit Legacy SSE Version)

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

VADDSUBPD (VEX.128 Encoded Version)

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

VADDSUBPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0] - SRC2[63:0]
DEST[127:64] := SRC1[127:64] + SRC2[127:64]
DEST[191:128] := SRC1[191:128] - SRC2[191:128]
DEST[255:192] := SRC1[255:192] + SRC2[255:192]
```

Intel C/C++ Compiler Intrinsic Equivalent

ADDSUBPD __m128d _mm_addsub_pd(__m128d a, __m128d b)
VADDSUBPD __m256d _mm256_addsub_pd (__m256d a, __m256d b)

Exceptions

When the source operand is a memory operand, it must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Table 2-19, "Type 2 Class Exception Conditions."

ADDSUBPS—Packed Single Precision Floating-Point Add/Subtract

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

Instruction Operand Encoding

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

Description

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

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

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified. See Figure 3-4.

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

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

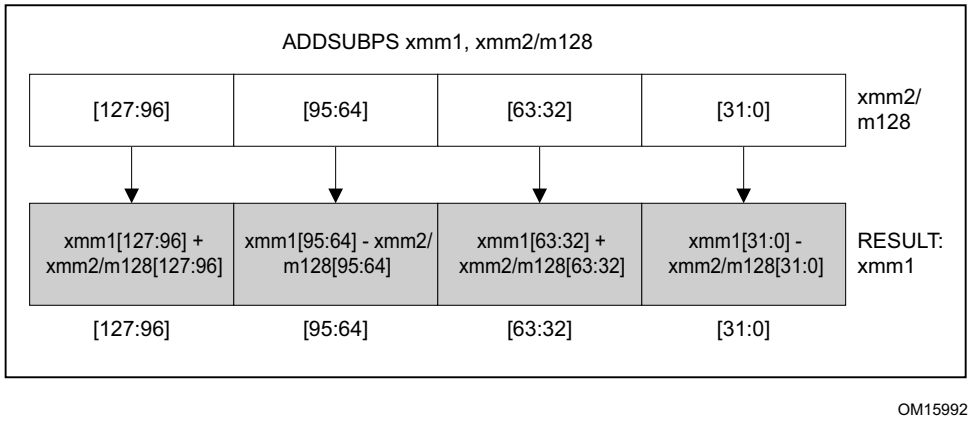


Figure 3-4. ADDSUBPS—Packed Single Precision Floating-Point Add/Subtract

Operation

ADDSUBPS (128-bit Legacy SSE Version)

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

VADDSUBPS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] - SRC2[31:0]
DEST[63:32] := SRC1[63:32] + SRC2[63:32]
DEST[95:64] := SRC1[95:64] - SRC2[95:64]
DEST[127:96] := SRC1[127:96] + SRC2[127:96]
DEST[MAXVL-1:128] := 0
```

VADDSUBPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[31:0] - SRC2[31:0]
DEST[63:32] := SRC1[63:32] + SRC2[63:32]
DEST[95:64] := SRC1[95:64] - SRC2[95:64]
DEST[127:96] := SRC1[127:96] + SRC2[127:96]
DEST[159:128] := SRC1[159:128] - SRC2[159:128]
DEST[191:160] := SRC1[191:160] + SRC2[191:160]
DEST[223:192] := SRC1[223:192] - SRC2[223:192]
DEST[255:224] := SRC1[255:224] + SRC2[255:224]
```

Intel C/C++ Compiler Intrinsic Equivalent

ADDSUBPS __m128 _mm_addsub_ps(__m128 a, __m128 b)
VADDSUBPS __m256 _mm256_addsub_ps (__m256 a, __m256 b)

Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Table 2-19, "Type 2 Class Exception Conditions."

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	RM	V/V	ADX	Unsigned addition of r32 with OF, r/m32 to r32, writes OF.
F3 REX.w 0F 38 F6 /r ADOX r64, r/m64	RM	V/N.E.	ADX	Unsigned addition of r64 with OF, r/m64 to r64, writes OF.

Instruction Operand Encoding

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

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.

Protected Mode Exceptions

#UD	If the LOCK prefix is used. If CPUID.07H.00H:EBX.ADX[19] = 0.
#SS(0)	For an illegal address in the SS segment.
#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used. If CPUID.07H.00H:EBX.ADX[19] = 0.
#SS(0)	For an illegal address in the SS segment.
#GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used. If CPUID.07H.00H:EBX.ADX[19] = 0.
#SS(0)	For an illegal address in the SS segment.
#GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used. If CPUID.07H.00H:EBX.ADX[19] = 0.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

AESDEC—Perform One Round of an AES Decryption Flow

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DE /r AESDEC xmm1, xmm2/m128	A	V/V	AES	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm1 with one 128-bit round key from xmm2/m128.
VEX.128.66.0F38.WIG DE /r VAESDEC xmm1, xmm2, xmm3/m128	B	V/V	AES AVX	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.
VEX.256.66.0F38.WIG DE /r VAESDEC ymm1, ymm2, ymm3/m256	B	V/V	VAES	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.
EVEX.128.66.0F38.WIG DE /r VAESDEC xmm1, xmm2, xmm3/m128	C	V/V	VAES (AVX512VL OR AVX10.1)	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.
EVEX.256.66.0F38.WIG DE /r VAESDEC ymm1, ymm2, ymm3/m256	C	V/V	VAES (AVX512VL OR AVX10.1)	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.
EVEX.512.66.0F38.WIG DE /r VAESDEC zmm1, zmm2, zmm3/m512	C	V/V	VAES (AVX512F OR AVX10.1)	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using four 128-bit data (state) from zmm2 with four 128-bit round keys from zmm3/m512; store the result in zmm1.

Instruction Operand Encoding

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

Description

This instruction performs a single round of the AES decryption flow using the Equivalent Inverse Cipher, using one/two/four (depending on vector length) 128-bit data (state) from the first source operand with one/two/four (depending on vector length) round key(s) from the second source operand, and stores the result in the destination operand.

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

VEX and EVEX encoded versions of the instruction allow 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.

Operation

AESDEC

```
STATE := SRC1;
RoundKey := SRC2;
STATE := InvShiftRows( STATE );
STATE := InvSubBytes( STATE );
STATE := InvMixColumns( STATE );
DEST[127:0] := STATE XOR RoundKey;
DEST[MAXVL-1:128] (Unmodified)
```

VAESDEC (128b and 256b VEX Encoded Versions)

```
(KL,VL) = (1,128), (2,256)
FOR i = 0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := InvShiftRows( STATE )
    STATE := InvSubBytes( STATE )
    STATE := InvMixColumns( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0
```

VAESDEC (EVEX Encoded Version)

```
(KL,VL) = (1,128), (2,256), (4,512)
FOR i = 0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := InvShiftRows( STATE )
    STATE := InvSubBytes( STATE )
    STATE := InvMixColumns( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] :=0
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESDEC __m128i _mm_aesdec (__m128i, __m128i)
VAESDEC __m256i _mm256_aesdec_epi128(__m256i, __m256i);
VAESDEC __m512i _mm512_aesdec_epi128(__m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded: See Table 2-52, "Type E4NF Class Exception Conditions."

AESDEC128KL—Perform Ten Rounds of AES Decryption Flow With Key Locker Using 128-Bit Key

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 38 DD !(11):rrr:bbb AESDEC128KL xmm, m384	A	V/V	AESKLE	Decrypt xmm using 128-bit AES key indicated by handle at m384 and store result in xmm.

Instruction Operand Encoding

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

Description

The AESDEC128KL[1] instruction performs 10 rounds of AES to decrypt the first operand using the 128-bit key indicated by the handle from the second operand. It stores the result in the first operand if the operation succeeds (e.g., does not run into a handle violation failure).

Operation

AESDEC128KL

```
Handle := UnalignedLoad of 384 bit (SRC);       // Load is not guaranteed to be atomic.
Illegal Handle = (HandleReservedBitSet (Handle) ||
                  (Handle[0] AND (CPL > 0)) ||
                  Handle [2] ||
                  HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES128);
IF (Illegal Handle) {
    THEN RFLAGS.ZF := 1;
    ELSE
        (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate384 (Handle[383:0], IWKey);
        IF (Authentic == 0)
            THEN RFLAGS.ZF := 1;
            ELSE
                DEST := AES128Decrypt (DEST, UnwrappedKey) ;
                RFLAGS.ZF := 0;
        FI;
FI;
RFLAGS.OF, SF, AF, PF, CF := 0;
```

Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

Intel C/C++ Compiler Intrinsic Equivalent

AESDEC128KL unsigned char _mm_aesdec128kl_u8(__m128i* odata, __m128i idata, const void* h);

1. Further details on Key Locker and usage of this instruction can be found here: https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.
	If CPUID.07H.00H:ECX.KEY_LOCKER[23] = 0.
	If CR4.KL = 0.
	If CPUID.19H:EBX.AESKLE[0] = 0.
	If CR0.EM = 1.
	If CR4.OSFXSR = 0.
#NM	If CR0.TS = 1.
#PF	If a page fault occurs.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If the memory address is in a non-canonical form.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
	If a memory address referencing the SS segment is in a non-canonical form.

AESDEC256KL—Perform 14 Rounds of AES Decryption Flow With Key Locker Using 256-Bit Key

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 38 DF !(11):rrr:bbb AESDEC256KL xmm, m512	A	V/V	AESKLE	Decrypt xmm using 256-bit AES key indicated by handle at m512 and store result in xmm.

Instruction Operand Encoding

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

Description

The AESDEC256KL[1] instruction performs 14 rounds of AES to decrypt the first operand using the 256-bit key indicated by the handle from the second operand. It stores the result in the first operand if the operation succeeds (e.g., does not run into a handle violation failure).

Operation

AESDEC256KL

```
Handle := UnalignedLoad of 512 bit (SRC);       // Load is not guaranteed to be atomic.
Illegal Handle = (HandleReservedBitSet (Handle) ||
                  (Handle[0] AND (CPL > 0)) ||
                  Handle [2] ||
                  HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES256);
IF (Illegal Handle)
    THEN RFLAGS.ZF := 1;
    ELSE
        (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate512 (Handle[511:0], IWKey);
        IF (Authentic == 0)
            THEN RFLAGS.ZF := 1;
            ELSE
                DEST := AES256Decrypt (DEST, UnwrappedKey) ;
                RFLAGS.ZF := 0;
        FI;
FI;
RFLAGS.OF, SF, AF, PF, CF := 0;
```

Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

Intel C/C++ Compiler Intrinsic Equivalent

AESDEC256KL unsigned char _mm_aesdec256kl_u8(__m128i* odata, __m128i idata, const void* h);

1. Further details on Key Locker and usage of this instruction can be found here: https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.
	If CPUID.07H.00H:ECX.KEY_LOCKER[23] = 0.
	If CR4.KL = 0.
	If CPUID.19H:EBX.AESKLE[0] = 0.
	If CR0.EM = 1.
	If CR4.OSFXSR = 0.
#NM	If CR0.TS = 1.
#PF	If a page fault occurs.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If the memory address is in a non-canonical form.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
	If a memory address referencing the SS segment is in a non-canonical form.

AESDECLAST—Perform Last Round of an AES Decryption Flow

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DF /r AESDECLAST xmm1, xmm2/m128	A	V/V	AES	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm1 with one 128-bit round key from xmm2/m128.
VEX.128.66.0F38.WIG DF /r VAESDECLAST xmm1, xmm2, xmm3/m128	B	V/V	AES AVX	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.
VEX.256.66.0F38.WIG DF /r VAESDECLAST ymm1, ymm2, ymm3/m256	B	V/V	VAES	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.
EVEX.128.66.0F38.WIG DF /r VAESDECLAST xmm1, xmm2, xmm3/m128	C	V/V	VAES (AVX512VL OR AVX10.1)	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.
EVEX.256.66.0F38.WIG DF /r VAESDECLAST ymm1, ymm2, ymm3/m256	C	V/V	VAES (AVX512VL OR AVX10.1)	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.
EVEX.512.66.0F38.WIG DF /r VAESDECLAST zmm1, zmm2, zmm3/m512	C	V/V	VAES (AVX512F OR AVX10.1)	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using four128-bit data (state) from zmm2 with four 128-bit round keys from zmm3/m512; store the result in zmm1.

Instruction Operand Encoding

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

Description

This instruction performs the last round of the AES decryption flow using the Equivalent Inverse Cipher, using one/two/four (depending on vector length) 128-bit data (state) from the first source operand with one/two/four (depending on vector length) round key(s) from the second source operand, and stores the result in the destination operand.

VEX and EVEX encoded versions of the instruction allow 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.

Operation

AESDECLAST

```
STATE := SRC1;
RoundKey := SRC2;
STATE := InvShiftRows( STATE );
STATE := InvSubBytes( STATE );
DEST[127:0] := STATE XOR RoundKey;
DEST[MAXVL-1:128] (Unmodified)
```

VAESDECLAST (128b and 256b VEX Encoded Versions)

```
(KL,VL) = (1,128), (2,256)
FOR i = 0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := InvShiftRows( STATE )
    STATE := InvSubBytes( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0
```

VAESDECLAST (EVEX Encoded Version)

```
(KL,VL) = (1,128), (2,256), (4,512)
FOR i = 0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := InvShiftRows( STATE )
    STATE := InvSubBytes( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESDECLAST __m128i _mm_aesdeclast (__m128i, __m128i)
VAESDECLAST __m256i _mm256_aesdeclast_epi128(__m256i, __m256i);
VAESDECLAST __m512i _mm512_aesdeclast_epi128(__m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded: See Table 2-52, "Type E4NF Class Exception Conditions."

AESDECWIDE128KL—Perform Ten Rounds of AES Decryption Flow With Key Locker on 8 Blocks Using 128-Bit Key

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 38 D8 !(11):001:bbb AESDECWIDE128KL m384, <XMM0-7>	A	V/V		Decrypt XMM0-7 using 128-bit AES key indicated by handle at m384 and store each resultant block back to its corresponding register.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operands 2—9
A	N/A	ModRM:r/m (r)	Implicit XMM0-7 (r, w)

Description

The AESDECWIDE128KL[1] instruction performs ten rounds of AES to decrypt each of the eight blocks in XMM0-7 using the 128-bit key indicated by the handle from the second operand. It replaces each input block in XMM0-7 with its corresponding decrypted block if the operation succeeds (e.g., does not run into a handle violation failure).

Operation

AESDECWIDE128KL

```
Handle := UnalignedLoad of 384 bit (SRC);       // Load is not guaranteed to be atomic.
Illegal Handle = (HandleReservedBitSet (Handle) ||
                  (Handle[0] AND (CPL > 0)) ||
                  Handle [2] ||
                  HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES128);
IF (Illegal Handle)
    THEN RFLAGS.ZF := 1;
    ELSE
        (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate384 (Handle[383:0], IWKey);
        IF Authentic == 0 {
            THEN RFLAGS.ZF := 1;
            ELSE
                XMM0 := AES128Decrypt (XMM0, UnwrappedKey) ;
                XMM1 := AES128Decrypt (XMM1, UnwrappedKey) ;
                XMM2 := AES128Decrypt (XMM2, UnwrappedKey) ;
                XMM3 := AES128Decrypt (XMM3, UnwrappedKey) ;
                XMM4 := AES128Decrypt (XMM4, UnwrappedKey) ;
                XMM5 := AES128Decrypt (XMM5, UnwrappedKey) ;
                XMM6 := AES128Decrypt (XMM6, UnwrappedKey) ;
                XMM7 := AES128Decrypt (XMM7, UnwrappedKey) ;
                RFLAGS.ZF := 0;
        FI;
FI;
RFLAGS.OF, SF, AF, PF, CF := 0;
```

Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

1. Further details on Key Locker and usage of this instruction can be found here: https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html.

Intel C/C++ Compiler Intrinsic Equivalent

AESDECWIDE128KLunsigned char _mm_aesdecwide128kl_u8(__m128i odata[8], const __m128i idata[8], const void* h);

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.
	If CPUID.07H.00H:ECX.KEY_LOCKER[23] = 0.
	If CR4.KL = 0.
	If CPUID.19H:EBX.AESKLE[0] = 0.
	If CR0.EM = 1.
	If CR4.OSFXSR = 0.
	If CPUID.19H:EBX.AES_WIDE[2] = 0.
#NM	If CR0.TS = 1.
#PF	If a page fault occurs.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If the memory address is in a non-canonical form.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
	If a memory address referencing the SS segment is in a non-canonical form.

AESDECWIDE256KL—Perform 14 Rounds of AES Decryption Flow With Key Locker on 8 Blocks Using 256-Bit Key

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 38 D8 !(11):011:bbb AESDECWIDE256KL m512, <XMM0-7>	A	V/V	AES_WIDE	Decrypt XMM0-7 using 256-bit AES key indicated by handle at m512 and store each resultant block back to its corresponding register.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operands 2—9
A	N/A	ModRM:r/m (r)	Implicit XMM0-7 (r, w)

Description

The AESDECWIDE256KL[1] instruction performs 14 rounds of AES to decrypt each of the eight blocks in XMM0-7 using the 256-bit key indicated by the handle from the second operand. It replaces each input block in XMM0-7 with its corresponding decrypted block if the operation succeeds (e.g., does not run into a handle violation failure).

Operation

AESDECWIDE256KL

```
Handle := UnalignedLoad of 512 bit (SRC);       // Load is not guaranteed to be atomic.
Illegal Handle = (HandleReservedBitSet (Handle) ||
                  (Handle[0] AND (CPL > 0)) ||
                  Handle [2] ||
                  HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES256);
IF (Illegal Handle) {
    THEN RFLAGS.ZF := 1;
    ELSE
        (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate512 (Handle[511:0], IWKey);
        IF (Authentic == 0)
            THEN RFLAGS.ZF := 1;
            ELSE
                XMM0 := AES256Decrypt (XMM0, UnwrappedKey) ;
                XMM1 := AES256Decrypt (XMM1, UnwrappedKey) ;
                XMM2 := AES256Decrypt (XMM2, UnwrappedKey) ;
                XMM3 := AES256Decrypt (XMM3, UnwrappedKey) ;
                XMM4 := AES256Decrypt (XMM4, UnwrappedKey) ;
                XMM5 := AES256Decrypt (XMM5, UnwrappedKey) ;
                XMM6 := AES256Decrypt (XMM6, UnwrappedKey) ;
                XMM7 := AES256Decrypt (XMM7, UnwrappedKey) ;
                RFLAGS.ZF := 0;
        FI;
FI;
RFLAGS.OF, SF, AF, PF, CF := 0;
```

Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

1. Further details on Key Locker and usage of this instruction can be found here: https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html.

Intel C/C++ Compiler Intrinsic Equivalent

AESDECWIDE256KLunsigned char _mm_aesdecwide256kl_u8(__m128i odata[8], const __m128i idata[8], const void* h);

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.
	If CPUID.07H.00H:ECX.KEY_LOCKER[23] = 0.
	If CR4.KL = 0.
	If CPUID.19H:EBX.AESKLE[0] = 0.
	If CR0.EM = 1.
	If CR4.OSFXSR = 0.
	If CPUID.19H:EBX.AES_WIDE[2] = 0.
#NM	If CR0.TS = 1.
#PF	If a page fault occurs.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If the memory address is in a non-canonical form.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
	If a memory address referencing the SS segment is in a non-canonical form.

AESENC—Perform One Round of an AES Encryption Flow

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DC /r AESENC xmm1, xmm2/m128	A	V/V	AES	Perform one round of an AES encryption flow, using one 128-bit data (state) from xmm1 with one 128-bit round key from xmm2/m128.
VEX.128.66.0F38.WIG DC /r VAESENC xmm1, xmm2, xmm3/m128	B	V/V	AES AVX	Perform one round of an AES encryption flow, using one 128-bit data (state) from xmm2 with one 128-bit round key from the xmm3/m128; store the result in xmm1.
VEX.256.66.0F38.WIG DC /r VAESENC ymm1, ymm2, ymm3/m256	B	V/V	VAES	Perform one round of an AES encryption flow, using two 128-bit data (state) from ymm2 with two 128-bit round keys from the ymm3/m256; store the result in ymm1.
EVEX.128.66.0F38.WIG DC /r VAESENC xmm1, xmm2, xmm3/m128	C	V/V	VAES (AVX512VL OR AVX10.1)	Perform one round of an AES encryption flow, using one 128-bit data (state) from xmm2 with one 128-bit round key from the xmm3/m128; store the result in xmm1.
EVEX.256.66.0F38.WIG DC /r VAESENC ymm1, ymm2, ymm3/m256	C	V/V	VAES (AVX512VL OR AVX10.1)	Perform one round of an AES encryption flow, using two 128-bit data (state) from ymm2 with two 128-bit round keys from the ymm3/m256; store the result in ymm1.
EVEX.512.66.0F38.WIG DC /r VAESENC zmm1, zmm2, zmm3/m512	C	V/V	VAES (AVX512F OR AVX10.1)	Perform one round of an AES encryption flow, using four 128-bit data (state) from zmm2 with four 128-bit round keys from the zmm3/m512; store the result in zmm1.

Instruction Operand Encoding

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

Description

This instruction performs a single round of an AES encryption flow using one/two/four (depending on vector length) 128-bit data (state) from the first source operand with one/two/four (depending on vector length) round key(s) from the second source operand, and stores the result in the destination operand.

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

VEX and EVEX encoded versions of the instruction allow 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.

Operation

AESENC

```
STATE := SRC1;
RoundKey := SRC2;
STATE := ShiftRows( STATE );
STATE := SubBytes( STATE );
STATE := MixColumns( STATE );
DEST[127:0] := STATE XOR RoundKey;
DEST[MAXVL-1:128] (Unmodified)
```

VAESENC (128b and 256b VEX Encoded Versions)

```
(KL,VL) = (1,128), (2,256)
FOR I := 0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := ShiftRows( STATE )
    STATE := SubBytes( STATE )
    STATE := MixColumns( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0
```

VAESENC (EVEX Encoded Version)

```
(KL,VL) = (1,128), (2,256), (4,512)
FOR i := 0 to KL-1:
    STATE := SRC1.xmm[i] // xmm[i] is the i'th xmm word in the SIMD register
    RoundKey := SRC2.xmm[i]
    STATE := ShiftRows( STATE )
    STATE := SubBytes( STATE )
    STATE := MixColumns( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESENC __m128i _mm_aesenc (__m128i, __m128i)
VAESENC __m256i _mm256_aesenc_epi128(__m256i, __m256i);
VAESENC __m512i _mm512_aesenc_epi128(__m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded: See Table 2-52, "Type E4NF Class Exception Conditions."

AESENC128KL—Perform Ten Rounds of AES Encryption Flow With Key Locker Using 128-Bit Key

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 38 DC !(11):rrr:bbb AESENC128KL xmm, m384	A	V/V	AESKLE	Encrypt xmm using 128-bit AES key indicated by handle at m384 and store result in xmm.

Instruction Operand Encoding

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

Description

The AESENC128KL[1] instruction performs ten rounds of AES to encrypt the first operand using the 128-bit key indicated by the handle from the second operand. It stores the result in the first operand if the operation succeeds (e.g., does not run into a handle violation failure).

Operation

AESENC128KL

```
Handle := UnalignedLoad of 384 bit (SRC);       // Load is not guaranteed to be atomic.
Illegal Handle = (
                    HandleReservedBitSet (Handle) ||
                    (Handle[0] AND (CPL > 0)) ||
                    Handle [1] ||
                    HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES128
                    );
IF (Illegal Handle) {
    THEN RFLAGS.ZF := 1;
    ELSE
        (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate384 (Handle[383:0], IWKey);
        IF (Authentic == 0)
        THEN RFLAGS.ZF := 1;
        ELSE
            DEST := AES128Encrypt (DEST, UnwrappedKey) ;
            RFLAGS.ZF := 0;
        FI;
FI;
RFLAGS.OF, SF, AF, PF, CF := 0;
```

Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

Intel C/C++ Compiler Intrinsic Equivalent

AESENC128KL unsigned char _mm_aesenc128kl_u8(__m128i* odata, __m128i idata, const void* h);

1. Further details on Key Locker and usage of this instruction can be found here: https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.
	If CPUID.07H.00H:ECX.KEY_LOCKER[23] = 0.
	If CR4.KL = 0.
	If CPUID.19H:EBX.AESKLE[0] = 0.
	If CR0.EM = 1.
	If CR4.OSFXSR = 0.
#NM	If CR0.TS = 1.
#PF	If a page fault occurs.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If the memory address is in a non-canonical form.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
	If a memory address referencing the SS segment is in a non-canonical form.

AESENC256KL—Perform 14 Rounds of AES Encryption Flow With Key Locker Using 256-Bit Key

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 38 DE !(11):rrr:bbb AESENC256KL xmm, m512	A	V/V	AESKLE	Encrypt xmm using 256-bit AES key indicated by handle at m512 and store result in xmm.

Instruction Operand Encoding

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

Description

The AESENC256KL[1] instruction performs 14 rounds of AES to encrypt the first operand using the 256-bit key indicated by the handle from the second operand. It stores the result in the first operand if the operation succeeds (e.g., does not run into a handle violation failure).

Operation

AESENC256KL

```
Handle := UnalignedLoad of 512 bit (SRC);       // Load is not guaranteed to be atomic.
Illegal Handle = (
                HandleReservedBitSet (Handle) ||
                (Handle[0] AND (CPL > 0)) ||
                Handle [1] ||
                HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES256
                );
IF (Illegal Handle)
    THEN RFLAGS.ZF := 1;
    ELSE
        (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate512 (Handle[511:0], IWKey);
        IF (Authentic == 0)
            THEN RFLAGS.ZF := 1;
            ELSE
                DEST := AES256Encrypt (DEST, UnwrappedKey) ;
                RFLAGS.ZF := 0;
        FI;
FI;
RFLAGS.OF, SF, AF, PF, CF := 0;
```

Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

Intel C/C++ Compiler Intrinsic Equivalent

AESENC256KL unsigned char _mm_aesenc256kl_u8(__m128i* odata, __m128i idata, const void* h);

1. Further details on Key Locker and usage of this instruction can be found here: https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.
	If CPUID.07H.00H:ECX.KEY_LOCKER[23] = 0.
	If CR4.KL = 0.
	If CPUID.19H:EBX.AESKLE[0] = 0.
	If CR0.EM = 1.
	If CR4.OSFXSR = 0.
#NM	If CR0.TS = 1.
#PF	If a page fault occurs.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If the memory address is in a non-canonical form.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
	If a memory address referencing the SS segment is in a non-canonical form.

AESENCLAST—Perform Last Round of an AES Encryption Flow

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DD /r AESENCLAST xmm1, xmm2/m128	A	V/V	AES	Perform the last round of an AES encryption flow, using one 128-bit data (state) from xmm1 with one 128-bit round key from xmm2/m128.
VEX.128.66.0F38.WIG DD /r VAESENCLAST xmm1, xmm2, xmm3/m128	B	V/V	AES AVX	Perform the last round of an AES encryption flow, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.
VEX.256.66.0F38.WIG DD /r VAESENCLAST ymm1, ymm2, ymm3/m256	B	V/V	VAES	Perform the last round of an AES encryption flow, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.
EVEX.128.66.0F38.WIG DD /r VAESENCLAST xmm1, xmm2, xmm3/m128	C	V/V	VAES (AVX512VL OR AVX10.1)	Perform the last round of an AES encryption flow, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.
EVEX.256.66.0F38.WIG DD /r VAESENCLAST ymm1, ymm2, ymm3/m256	C	V/V	VAES (AVX512VL OR AVX10.1)	Perform the last round of an AES encryption flow, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.
EVEX.512.66.0F38.WIG DD /r VAESENCLAST zmm1, zmm2, zmm3/m512	C	V/V	VAES (AVX512F OR AVX10.1)	Perform the last round of an AES encryption flow, using four 128-bit data (state) from zmm2 with four 128-bit round keys from zmm3/m512; store the result in zmm1.

Instruction Operand Encoding

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

Description

This instruction performs the last round of an AES encryption flow using one/two/four (depending on vector length) 128-bit data (state) from the first source operand with one/two/four (depending on vector length) round key(s) from the second source operand, and stores the result in the destination operand.

VEX and EVEX encoded versions of the instruction allows 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.

Operation

AESENCLAST

```
STATE := SRC1;
RoundKey := SRC2;
STATE := ShiftRows( STATE );
STATE := SubBytes( STATE );
DEST[127:0] := STATE XOR RoundKey;
DEST[MAXVL-1:128] (Unmodified)
```

VAESENCLAST (128b and 256b VEX Encoded Versions)

```
(KL, VL) = (1,128), (2,256)
FOR I=0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := ShiftRows( STATE )
    STATE := SubBytes( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0
```

VAESENCLAST (EVEX Encoded Version)

```
(KL,VL) = (1,128), (2,256), (4,512)
FOR i = 0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := ShiftRows( STATE )
    STATE := SubBytes( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESENCLAST __m128i _mm_aesenclast (__m128i, __m128i)
VAESENCLAST __m256i _mm256_aesenclast_epi128(__m256i, __m256i);
VAESENCLAST __m512i _mm512_aesenclast_epi128(__m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded: See Table 2-52, "Type E4NF Class Exception Conditions."

AESENCWIDE128KL—Perform Ten Rounds of AES Encryption Flow With Key Locker on 8 Blocks Using 128-Bit Key

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 38 D8 !(11):000:bbb AESENCWIDE128KL m384, <XMM0-7>	A	V/V	AES_WIDE	Encrypt XMM0-7 using 128-bit AES key indicated by handle at m384 and store each resultant block back to its corresponding register.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operands 2—9
A	N/A	ModRM:r/m (r)	Implicit XMM0-7 (r, w)

Description

The AESENCWIDE128KL[1] instruction performs ten rounds of AES to encrypt each of the eight blocks in XMM0-7 using the 128-bit key indicated by the handle from the second operand. It replaces each input block in XMM0-7 with its corresponding encrypted block if the operation succeeds (e.g., does not run into a handle violation failure).

Operation

AESENCWIDE128KL

```
Handle := UnalignedLoad of 384 bit (SRC);       // Load is not guaranteed to be atomic.
Illegal Handle = (
                HandleReservedBitSet (Handle) ||
                (Handle[0] AND (CPL > 0)) ||
                Handle [1] ||
                HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES128
                );
IF (Illegal Handle)
    THEN RFLAGS.ZF := 1;
    ELSE
        (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate384 (Handle[383:0], IWKey);
        IF Authentic == 0
            THEN RFLAGS.ZF := 1;
            ELSE
            XMM0 := AES128Encrypt (XMM0, UnwrappedKey) ;
                XMM1 := AES128Encrypt (XMM1, UnwrappedKey) ;
                XMM2 := AES128Encrypt (XMM2, UnwrappedKey) ;
                XMM3 := AES128Encrypt (XMM3, UnwrappedKey) ;
                XMM4 := AES128Encrypt (XMM4, UnwrappedKey) ;
                XMM5 := AES128Encrypt (XMM5, UnwrappedKey) ;
                XMM6 := AES128Encrypt (XMM6, UnwrappedKey) ;
                XMM7 := AES128Encrypt (XMM7, UnwrappedKey) ;
                RFLAGS.ZF := 0;
        FI;
FI;
RFLAGS.OF, SF, AF, PF, CF := 0;
```

1. Further details on Key Locker and usage of this instruction can be found here: https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html.

Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

Intel C/C++ Compiler Intrinsic Equivalent

AESENCWIDE128KLunsigned char _mm_aesencwide128kl_u8(__m128i odata[8], const __m128i idata[8], const void* h);

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.
	If CPUID.07H.00H:ECX.KEY_LOCKER[23] = 0.
	If CR4.KL = 0.
	If CPUID.19H:EBX.AESKLE[0] = 0.
	If CR0.EM = 1.
	If CR4.OSFXSR = 0.
	If CPUID.19H:EBX.AES_WIDE[2] = 0.
#NM	If CR0.TS = 1.
#PF	If a page fault occurs.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If the memory address is in a non-canonical form.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
	If a memory address referencing the SS segment is in a non-canonical form.

AESENCWIDE256KL—Perform 14 Rounds of AES Encryption Flow With Key Locker on 8 Blocks Using 256-Bit Key

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 38 D8 !(11):010:bbb AESENCWIDE256KL m512, <XMM0-7>	A	V/V	AES_WIDE	Encrypt XMM0-7 using 256-bit AES key indicated by handle at m512 and store each resultant block back to its corresponding register.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operands 2—9
A	N/A	ModRM:r/m (r)	Implicit XMM0-7 (r, w)

Description

The AESENCWIDE256KL[1] instruction performs 14 rounds of AES to encrypt each of the eight blocks in XMM0-7 using the 256-bit key indicated by the handle from the second operand. It replaces each input block in XMM0-7 with its corresponding encrypted block if the operation succeeds (e.g., does not run into a handle violation failure).

Operation

AESENCWIDE256KL

```
Handle := UnalignedLoad of 512 bit (SRC);       // Load is not guaranteed to be atomic.
Illegal Handle = (
                HandleReservedBitSet (Handle) ||
                (Handle[0] AND (CPL > 0)) ||
                Handle [1] ||
                HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES256
                );
IF (Illegal Handle)
    THEN RFLAGS.ZF := 1;
    ELSE
        (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate512 (Handle[511:0], IWKey);
        IF (Authentic == 0)
            THEN RFLAGS.ZF := 1;
            ELSE
                XMM0 := AES256Encrypt (XMM0, UnwrappedKey) ;
                XMM1 := AES256Encrypt (XMM1, UnwrappedKey) ;
                XMM2 := AES256Encrypt (XMM2, UnwrappedKey) ;
                XMM3 := AES256Encrypt (XMM3, UnwrappedKey) ;
                XMM4 := AES256Encrypt (XMM4, UnwrappedKey) ;
                XMM5 := AES256Encrypt (XMM5, UnwrappedKey) ;
                XMM6 := AES256Encrypt (XMM6, UnwrappedKey) ;
                XMM7 := AES256Encrypt (XMM7, UnwrappedKey) ;
                RFLAGS.ZF := 0;
        FI;
FI;
RFLAGS.OF, SF, AF, PF, CF := 0;
```

1. Further details on Key Locker and usage of this instruction can be found here: https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html.

Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

Intel C/C++ Compiler Intrinsic Equivalent

AESENCWIDE256KLunsigned char _mm_aesencwide256kl_u8(__m128i odata[8], const __m128i idata[8], const void* h);

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.
	If CPUID.07H.00H:ECX.KEY_LOCKER[23] = 0.
	If CR4.KL = 0.
	If CPUID.19H:EBX.AESKLE[0] = 0.
	If CR0.EM = 1.
	If CR4.OSFXSR = 0.
	If CPUID.19H:EBX.AES_WIDE[2] = 0.
#NM	If CR0.TS = 1.
#PF	If a page fault occurs.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If the memory address is in a non-canonical form.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
	If a memory address referencing the SS segment is in a non-canonical form.

AESIMC—Perform the AES InvMixColumn Transformation

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

Instruction Operand Encoding

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

Description

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

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

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

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

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

Operation

AESIMC

```
DEST[127:0] := InvMixColumns( SRC );
DEST[MAXVL-1:128] (Unmodified)
```

VAESIMC

```
DEST[127:0] := InvMixColumns( SRC );
DEST[MAXVL-1:128] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESIMC __m128i _mm_aesimc (__m128i)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions," additionally:

#UD	If VEX.vvvv ≠ 1111B.

AESKEYGENASSIST—AES Round Key Generation Assist

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

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

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

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

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

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

Operation

AESKEYGENASSIST

```
X3[31:0] := SRC [127: 96];
X2[31:0] := SRC [95: 64];
X1[31:0] := SRC [63: 32];
X0[31:0] := SRC [31: 0];
RCON[31:0] := ZeroExtend(imm8[7:0]);
DEST[31:0] := SubWord(X1);
DEST[63:32 ] := RotWord( SubWord(X1) ) XOR RCON;
DEST[95:64] := SubWord(X3);
DEST[127:96] := RotWord( SubWord(X3) ) XOR RCON;
DEST[MAXVL-1:128] (Unmodified)
```

VAESKEYGENASSIST

```
X3[31:0] := SRC [127: 96];
X2[31:0] := SRC [95: 64];
X1[31:0] := SRC [63: 32];
X0[31:0] := SRC [31: 0];
RCON[31:0] := ZeroExtend(imm8[7:0]);
DEST[31:0] := SubWord(X1);
DEST[63:32 ] := RotWord( SubWord(X1) ) XOR RCON;
DEST[95:64] := SubWord(X3);
DEST[127:96] := RotWord( SubWord(X3) ) XOR RCON;
DEST[MAXVL-1:128] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions," additionally:

#UD	If VEX.vvvv ≠ 1111B.

AND—Logical AND

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
24 ib	AND AL, imm8	I	Valid	Valid	AL AND imm8.
25 iw	AND AX, imm16	I	Valid	Valid	AX AND imm16.
25 id	AND EAX, imm32	I	Valid	Valid	EAX AND imm32.
REX.W + 25 id	AND RAX, imm32	I	Valid	N.E.	RAX AND imm32 sign-extended to 64-bits.
80 /4 ib	AND r/m8[1], imm8[1]	MI	Valid	Valid	r/m8 AND imm8.
81 /4 iw	AND r/m16, imm16	MI	Valid	Valid	r/m16 AND imm16.
81 /4 id	AND r/m32, imm32	MI	Valid	Valid	r/m32 AND imm32.
REX.W + 81 /4 id	AND r/m64, imm32	MI	Valid	N.E.	r/m64 AND imm32 sign extended to 64-bits.
83 /4 ib	AND r/m16, imm8	MI	Valid	Valid	r/m16 AND imm8 (sign-extended).
83 /4 ib	AND r/m32, imm8	MI	Valid	Valid	r/m32 AND imm8 (sign-extended).
REX.W + 83 /4 ib	AND r/m64, imm8	MI	Valid	N.E.	r/m64 AND imm8 (sign-extended).
20 /r	AND r/m8[1], r8[1]	MR	Valid	Valid	r/m8 AND r8.
21 /r	AND r/m16, r16	MR	Valid	Valid	r/m16 AND r16.
21 /r	AND r/m32, r32	MR	Valid	Valid	r/m32 AND r32.
REX.W + 21 /r	AND r/m64, r64	MR	Valid	N.E.	r/m64 AND r64.
22 /r	AND r8[1], r/m8[1]	RM	Valid	Valid	r8 AND r/m8.
23 /r	AND r16, r/m16	RM	Valid	Valid	r16 AND r/m16.
23 /r	AND r32, r/m32	RM	Valid	Valid	r32 AND r/m32.
REX.W + 23 /r	AND r64, r/m64	RM	Valid	N.E.	r64 AND r/m64.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
MR	ModRM:r/m (r, w)	ModRM:reg (r)	N/A	N/A
MI	ModRM:r/m (r, w)	imm8/16/32	N/A	N/A
I	AL/AX/EAX/RAX	imm8/16/32	N/A	N/A

Description

Performs a bitwise AND operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is set to 1 if both corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

This instruction can be used with a LOCK prefix to allow the it to be executed atomically.

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

Operation

```
DEST := DEST AND SRC;
```

Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

ANDN—Logical AND NOT

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.LZ.0F38.W0 F2 /r ANDN r32a, r32b, r/m32	RVM	V/V	BMI1	Bitwise AND of inverted r32b with r/m32, store result in r32a.
VEX.LZ. 0F38.W1 F2 /r ANDN r64a, r64b, r/m64	RVM	V/N.E.	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
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

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 2-29, "Type 13 Class Exception Conditions."

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 55 /r ANDNPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical AND NOT of packed double precision floating-point values in xmm1 and xmm2/mem.
VEX.128.66.0F 55 /r VANDNPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND NOT of packed double precision floating-point values in xmm2 and xmm3/mem.
VEX.256.66.0F 55/r VANDNPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND NOT of packed double precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.66.0F.W1 55 /r VANDNPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical AND NOT of packed double precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.256.66.0F.W1 55 /r VANDNPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical AND NOT of packed double precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.512.66.0F.W1 55 /r VANDNPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ OR AVX10.1	Return the bitwise logical AND NOT of packed double precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

Instruction Operand Encoding

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

Description

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

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

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

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

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

Operation

VANDNPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := (NOT(SRC1[i+63:i])) BITWISE AND SRC2[63:0]
                ELSE
                    DEST[i+63:i] := (NOT(SRC1[i+63:i])) BITWISE AND SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] = 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VANDNPD (VEX.256 Encoded Version)

```
DEST[63:0] := (NOT(SRC1[63:0])) BITWISE AND SRC2[63:0]
DEST[127:64] := (NOT(SRC1[127:64])) BITWISE AND SRC2[127:64]
DEST[191:128] := (NOT(SRC1[191:128])) BITWISE AND SRC2[191:128]
DEST[255:192] := (NOT(SRC1[255:192])) BITWISE AND SRC2[255:192]
DEST[MAXVL-1:256] := 0
```

VANDNPD (VEX.128 Encoded Version)

```
DEST[63:0] := (NOT(SRC1[63:0])) BITWISE AND SRC2[63:0]
DEST[127:64] := (NOT(SRC1[127:64])) BITWISE AND SRC2[127:64]
DEST[MAXVL-1:128] := 0
```

ANDNPD (128-bit Legacy SSE Version)

```
DEST[63:0] := (NOT(DEST[63:0])) BITWISE AND SRC[63:0]
DEST[127:64] := (NOT(DEST[127:64])) BITWISE AND SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VANDNPD __m512d _mm512_andnot_pd (__m512d a, __m512d b);
VANDNPD __m512d _mm512_mask_andnot_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);
VANDNPD __m512d _mm512_maskz_andnot_pd (__mmask8 k, __m512d a, __m512d b);
VANDNPD __m256d _mm256_mask_andnot_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);
VANDNPD __m256d _mm256_maskz_andnot_pd (__mmask8 k, __m256d a, __m256d b);
VANDNPD __m128d _mm_mask_andnot_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VANDNPD __m128d _mm_maskz_andnot_pd (__mmask8 k, __m128d a, __m128d b);
VANDNPD __m256d _mm256_andnot_pd (__m256d a, __m256d b);
ANDNPD __m128d _mm_andnot_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 55 /r ANDNPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical AND NOT of packed single precision floating-point values in xmm1 and xmm2/mem.
VEX.128.0F 55 /r VANDNPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND NOT of packed single precision floating-point values in xmm2 and xmm3/mem.
VEX.256.0F 55 /r VANDNPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND NOT of packed single precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.0F.W0 55 /r VANDNPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical AND of packed single precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.256.0F.W0 55 /r VANDNPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical AND of packed single precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.512.0F.W0 55 /r VANDNPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512DQ OR AVX10.1	Return the bitwise logical AND of packed single precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

Instruction Operand Encoding

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

Description

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

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

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

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

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

Operation

VANDNPS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] := (NOT(SRC1[i+31:i])) BITWISE AND SRC2[31:0]
                ELSE
                    DEST[i+31:i] := (NOT(SRC1[i+31:i])) BITWISE AND SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] = 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VANDNPS (VEX.256 Encoded Version)

```
DEST[31:0] := (NOT(SRC1[31:0])) BITWISE AND SRC2[31:0]
DEST[63:32] := (NOT(SRC1[63:32])) BITWISE AND SRC2[63:32]
DEST[95:64] := (NOT(SRC1[95:64])) BITWISE AND SRC2[95:64]
DEST[127:96] := (NOT(SRC1[127:96])) BITWISE AND SRC2[127:96]
DEST[159:128] := (NOT(SRC1[159:128])) BITWISE AND SRC2[159:128]
DEST[191:160] := (NOT(SRC1[191:160])) BITWISE AND SRC2[191:160]
DEST[223:192] := (NOT(SRC1[223:192])) BITWISE AND SRC2[223:192]
DEST[255:224] := (NOT(SRC1[255:224])) BITWISE AND SRC2[255:224].
DEST[MAXVL-1:256] := 0
```

VANDNPS (VEX.128 Encoded Version)

```
DEST[31:0] := (NOT(SRC1[31:0])) BITWISE AND SRC2[31:0]
DEST[63:32] := (NOT(SRC1[63:32])) BITWISE AND SRC2[63:32]
DEST[95:64] := (NOT(SRC1[95:64])) BITWISE AND SRC2[95:64]
DEST[127:96] := (NOT(SRC1[127:96])) BITWISE AND SRC2[127:96]
DEST[MAXVL-1:128] := 0
```

ANDNPS (128-bit Legacy SSE Version)

```
DEST[31:0] := (NOT(DEST[31:0])) BITWISE AND SRC[31:0]
DEST[63:32] := (NOT(DEST[63:32])) BITWISE AND SRC[63:32]
DEST[95:64] := (NOT(DEST[95:64])) BITWISE AND SRC[95:64]
DEST[127:96] := (NOT(DEST[127:96])) BITWISE AND SRC[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VANDNPS __m512 _mm512_andnot_ps (__m512 a, __m512 b);
VANDNPS __m512 _mm512_mask_andnot_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VANDNPS __m512 _mm512_maskz_andnot_ps (__mmask16 k, __m512 a, __m512 b);
VANDNPS __m256 _mm256_mask_andnot_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VANDNPS __m256 _mm256_maskz_andnot_ps (__mmask8 k, __m256 a, __m256 b);
VANDNPS __m128 _mm_mask_andnot_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VANDNPS __m128 _mm_maskz_andnot_ps (__mmask8 k, __m128 a, __m128 b);
VANDNPS __m256 _mm256_andnot_ps (__m256 a, __m256 b);
ANDNPS __m128 _mm_andnot_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."

EVEX-encoded instruction, see Table 2-51, "Type E4 Class Exception Conditions."

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 54 /r ANDPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical AND of packed double precision floating-point values in xmm1 and xmm2/mem.
VEX.128.66.0F 54 /r VANDPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND of packed double precision floating-point values in xmm2 and xmm3/mem.
VEX.256.66.0F 54 /r VANDPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND of packed double precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.66.0F.W1 54 /r VANDPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical AND of packed double precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.256.66.0F.W1 54 /r VANDPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical AND of packed double precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.512.66.0F.W1 54 /r VANDPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ OR AVX10.1	Return the bitwise logical AND of packed double precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

Instruction Operand Encoding

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

Description

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

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

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

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

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

Operation

VANDPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := SRC1[i+63:i] BITWISE AND SRC2[63:0]
                ELSE
                    DEST[i+63:i] := SRC1[i+63:i] BITWISE AND SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] = 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VANDPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0] BITWISE AND SRC2[63:0]
DEST[127:64] := SRC1[127:64] BITWISE AND SRC2[127:64]
DEST[191:128] := SRC1[191:128] BITWISE AND SRC2[191:128]
DEST[255:192] := SRC1[255:192] BITWISE AND SRC2[255:192]
DEST[MAXVL-1:256] := 0
```

VANDPD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] BITWISE AND SRC2[63:0]
DEST[127:64] := SRC1[127:64] BITWISE AND SRC2[127:64]
DEST[MAXVL-1:128] := 0
```

ANDPD (128-bit Legacy SSE Version)

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

Intel C/C++ Compiler Intrinsic Equivalent

VANDPD __m512d _mm512_and_pd (__m512d a, __m512d b);
VANDPD __m512d _mm512_mask_and_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);
VANDPD __m512d _mm512_maskz_and_pd (__mmask8 k, __m512d a, __m512d b);
VANDPD __m256d _mm256_mask_and_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);
VANDPD __m256d _mm256_maskz_and_pd (__mmask8 k, __m256d a, __m256d b);
VANDPD __m128d _mm_mask_and_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VANDPD __m128d _mm_maskz_and_pd (__mmask8 k, __m128d a, __m128d b);
VANDPD __m256d _mm256_and_pd (__m256d a, __m256d b);
ANDPD __m128d _mm_and_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 54 /r ANDPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical AND of packed single precision floating-point values in xmm1 and xmm2/mem.
VEX.128.0F 54 /r VANDPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND of packed single precision floating-point values in xmm2 and xmm3/mem.
VEX.256.0F 54 /r VANDPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND of packed single precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.0F.W0 54 /r VANDPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical AND of packed single precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.256.0F.W0 54 /r VANDPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical AND of packed single precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.512.0F.W0 54 /r VANDPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512DQ OR AVX10.1	Return the bitwise logical AND of packed single precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

Instruction Operand Encoding

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

Description

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

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

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

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

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

Operation

VANDPS (EVEX Encoded Versions)

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

VANDPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[31:0] BITWISE AND SRC2[31:0]
DEST[63:32] := SRC1[63:32] BITWISE AND SRC2[63:32]
DEST[95:64] := SRC1[95:64] BITWISE AND SRC2[95:64]
DEST[127:96] := SRC1[127:96] BITWISE AND SRC2[127:96]
DEST[159:128] := SRC1[159:128] BITWISE AND SRC2[159:128]
DEST[191:160] := SRC1[191:160] BITWISE AND SRC2[191:160]
DEST[223:192] := SRC1[223:192] BITWISE AND SRC2[223:192]
DEST[255:224] := SRC1[255:224] BITWISE AND SRC2[255:224].
DEST[MAXVL-1:256] := 0;
```

VANDPS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] BITWISE AND SRC2[31:0]
DEST[63:32] := SRC1[63:32] BITWISE AND SRC2[63:32]
DEST[95:64] := SRC1[95:64] BITWISE AND SRC2[95:64]
DEST[127:96] := SRC1[127:96] BITWISE AND SRC2[127:96]
DEST[MAXVL-1:128] := 0;
```

ANDPS (128-bit Legacy SSE Version)

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

Intel C/C++ Compiler Intrinsic Equivalent

VANDPS __m512 _mm512_and_ps (__m512 a, __m512 b);
VANDPS __m512 _mm512_mask_and_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VANDPS __m512 _mm512_maskz_and_ps (__mmask16 k, __m512 a, __m512 b);
VANDPS __m256 _mm256_mask_and_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VANDPS __m256 _mm256_maskz_and_ps (__mmask8 k, __m256 a, __m256 b);
VANDPS __m128 _mm_mask_and_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VANDPS __m128 _mm_maskz_and_ps (__mmask8 k, __m128 a, __m128 b);
VANDPS __m256 _mm256_and_ps (__m256 a, __m256 b);
ANDPS __m128 _mm_and_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."

EVEX-encoded instruction, see Table 2-51, "Type E4 Class Exception Conditions."

ARPL—Adjust RPL Field of Segment Selector

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
63 /r	ARPL r/m16, r16	MR	N. E.	Valid	Adjust RPL of r/m16 to not less than RPL of r16.

Instruction Operand Encoding

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

Description

Compares the RPL fields of two segment selectors. The first operand (the destination operand) contains one segment selector and the second operand (source operand) contains the other. (The RPL field is located in bits 0 and 1 of each operand.) If the RPL field of the destination operand is less than the RPL field of the source operand, the ZF flag is set and the RPL field of the destination operand is increased to match that of the source operand. Otherwise, the ZF flag is cleared and no change is made to the destination operand. (The destination operand can be a word register or a memory location; the source operand must be a word register.)

The ARPL instruction is provided for use by operating-system procedures (however, it can also be used by applications). It is generally used to adjust the RPL of a segment selector that has been passed to the operating system by an application program to match the privilege level of the application program. Here the segment selector passed to the operating system is placed in the destination operand and segment selector for the application program's code segment is placed in the source operand. (The RPL field in the source operand represents the privilege level of the application program.) Execution of the ARPL instruction then ensures that the RPL of the segment selector received by the operating system is no lower (does not have a higher privilege) than the privilege level of the application program (the segment selector for the application program's code segment can be read from the stack following a procedure call).

This instruction executes as described in compatibility mode and legacy mode. It is not encodable in 64-bit mode.

See "Checking Caller Access Privileges" in Chapter 3, "Protected-Mode Memory Management," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, for more information about the use of this instruction.

Operation

```
IF 64-BIT MODE
    THEN
        See MOVSXD;
    ELSE
        IF DEST[RPL] < SRC[RPL]
            THEN
                ZF := 1;
                DEST[RPL] := SRC[RPL];
            ELSE
                ZF := 0;
        FI;
FI;
```

Flags Affected

The ZF flag is set to 1 if the RPL field of the destination operand is less than that of the source operand; otherwise, it is set to 0.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#UD	The ARPL instruction is not recognized in real-address mode.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Not applicable.

BEXTR—Bit Field Extract

Opcode/Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.LZ.0F38.W0 F7 /r BEXTR r32a, r/m32, r32b	RMV	V/V	BMI1	Contiguous bitwise extract from r/m32 using r32b as control; store result in r32a.
VEX.LZ.0F38.W1 F7 /r BEXTR r64a, r/m64, r64b	RMV	V/N.E.	BMI1	Contiguous bitwise extract from r/m64 using r64b as control; store result in r64a.

Instruction Operand Encoding

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

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 second 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 2-29, "Type 13 Class Exception Conditions," additionally:

#UD	If VEX.W = 1.

BLENDPD—Blend Packed Double Precision Floating-Point Values

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8[3:0]

Description

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

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

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

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

Operation

BLENDPD (128-bit Legacy SSE Version)

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

VBLENDPD (VEX.128 Encoded Version)

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

VBLENDPD (VEX.256 Encoded Version)

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

Intel C/C++ Compiler Intrinsic Equivalent

BLENDPD __m128d _mm_blend_pd (__m128d v1, __m128d v2, const int mask);
VBLENDPD __m256d _mm256_blend_pd (__m256d a, __m256d b, const int mask);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions."

BLENDPS—Blend Packed Single Precision Floating-Point Values

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

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

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

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

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

Operation

BLENDPS (128-bit Legacy SSE Version)

```
IF (IMM8[0] = 0) THEN DEST[31:0] :=DEST[31:0]
        ELSE DEST [31:0] := SRC[31:0] FI
IF (IMM8[1] = 0) THEN DEST[63:32] := DEST[63:32]
        ELSE DEST [63:32] := SRC[63:32] FI
IF (IMM8[2] = 0) THEN DEST[95:64] := DEST[95:64]
        ELSE DEST [95:64] := SRC[95:64] FI
IF (IMM8[3] = 0) THEN DEST[127:96] := DEST[127:96]
        ELSE DEST [127:96] := SRC[127:96] FI
DEST[MAXVL-1:128] (Unmodified)
```

VBLENDPS (VEX.128 Encoded Version)

```
IF (IMM8[0] = 0) THEN DEST[31:0] :=SRC1[31:0]
        ELSE DEST [31:0] := SRC2[31:0] FI
IF (IMM8[1] = 0) THEN DEST[63:32] := SRC1[63:32]
        ELSE DEST [63:32] := SRC2[63:32] FI
IF (IMM8[2] = 0) THEN DEST[95:64] := SRC1[95:64]
        ELSE DEST [95:64] := SRC2[95:64] FI
IF (IMM8[3] = 0) THEN DEST[127:96] := SRC1[127:96]
        ELSE DEST [127:96] := SRC2[127:96] FI
DEST[MAXVL-1:128] := 0
```

VBLENDPS (VEX.256 Encoded Version)

```
IF (IMM8[0] = 0) THEN DEST[31:0] :=SRC1[31:0]
        ELSE DEST [31:0] := SRC2[31:0] FI
IF (IMM8[1] = 0) THEN DEST[63:32] := SRC1[63:32]
        ELSE DEST [63:32] := SRC2[63:32] FI
IF (IMM8[2] = 0) THEN DEST[95:64] := SRC1[95:64]
        ELSE DEST [95:64] := SRC2[95:64] FI
IF (IMM8[3] = 0) THEN DEST[127:96] := SRC1[127:96]
        ELSE DEST [127:96] := SRC2[127:96] FI
IF (IMM8[4] = 0) THEN DEST[159:128] := SRC1[159:128]
        ELSE DEST [159:128] := SRC2[159:128] FI
IF (IMM8[5] = 0) THEN DEST[191:160] := SRC1[191:160]
        ELSE DEST [191:160] := SRC2[191:160] FI
IF (IMM8[6] = 0) THEN DEST[223:192] := SRC1[223:192]
        ELSE DEST [223:192] := SRC2[223:192] FI
IF (IMM8[7] = 0) THEN DEST[255:224] := SRC1[255:224]
        ELSE DEST [255:224] := SRC2[255:224] FI.
```

Intel C/C++ Compiler Intrinsic Equivalent

BLENDPS __m128 _mm_blend_ps (__m128 v1, __m128 v2, const int mask);
VBLENDPS __m256 _mm256_blend_ps (__m256 a, __m256 b, const int mask);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions."

BLENDVPD—Variable Blend Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 15 /r BLENDVPD xmm1, xmm2/m128 , <XMM0>	RM0	V/V	SSE4_1	Select packed double precision floating-point values from xmm1 and xmm2 from mask specified in XMM0 and store the values in xmm1.
VEX.128.66.0F3A.W0 4B /r /is4 VBLENDVPD xmm1, xmm2, xmm3/m128, xmm4	RVMR	V/V	AVX	Conditionally copy double precision floating-point values from xmm2 or xmm3/m128 to xmm1, based on mask bits in the mask operand, xmm4.
VEX.256.66.0F3A.W0 4B /r /is4 VBLENDVPD ymm1, ymm2, ymm3/m256, ymm4	RVMR	V/V	AVX	Conditionally copy double precision floating-point values from ymm2 or ymm3/m256 to ymm1, based on mask bits in the mask operand, ymm4.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM0	ModRM:reg (r, w)	ModRM:r/m (r)	implicit XMM0	N/A
RVMR	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8[7:4]

Description

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

Each quadword element of the destination operand is copied from:

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

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

128-bit Legacy SSE version: The first source operand and the destination operand is the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute BLENDVPD with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand is an XMM register or 128-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (MAXVL-1:128) of the corresponding YMM register (destination register) are zeroed. VEX.W must be 0, otherwise, the instruction will #UD.

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

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

Operation

BLENDVPD (128-bit Legacy SSE Version)

```
MASK := XMM0
IF (MASK[63] = 0) THEN DEST[63:0] := DEST[63:0]
        ELSE DEST [63:0] := SRC[63:0] FI
IF (MASK[127] = 0) THEN DEST[127:64] := DEST[127:64]
        ELSE DEST [127:64] := SRC[127:64] FI
DEST[MAXVL-1:128] (Unmodified)
```

VBLENDVPD (VEX.128 Encoded Version)

```
MASK := SRC3
IF (MASK[63] = 0) THEN DEST[63:0] := SRC1[63:0]
        ELSE DEST [63:0] := SRC2[63:0] FI
IF (MASK[127] = 0) THEN DEST[127:64] := SRC1[127:64]
        ELSE DEST [127:64] := SRC2[127:64] FI
DEST[MAXVL-1:128] := 0
```

VBLENDVPD (VEX.256 Encoded Version)

```
MASK := SRC3
IF (MASK[63] = 0) THEN DEST[63:0] := SRC1[63:0]
        ELSE DEST [63:0] := SRC2[63:0] FI
IF (MASK[127] = 0) THEN DEST[127:64] := SRC1[127:64]
        ELSE DEST [127:64] := SRC2[127:64] FI
IF (MASK[191] = 0) THEN DEST[191:128] := SRC1[191:128]
        ELSE DEST [191:128] := SRC2[191:128] FI
IF (MASK[255] = 0) THEN DEST[255:192] := SRC1[255:192]
        ELSE DEST [255:192] := SRC2[255:192] FI
```

Intel C/C++ Compiler Intrinsic Equivalent

BLENDVPD __m128d _mm_blendv_pd(__m128d v1, __m128d v2, __m128d v3);
VBLENDVPD __m128 _mm_blendv_pd (__m128d a, __m128d b, __m128d mask);
VBLENDVPD __m256 _mm256_blendv_pd (__m256d a, __m256d b, __m256d mask);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions," additionally:

#UD	If VEX.W = 1.

BLENDVPS—Variable Blend Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 14 /r BLENDVPS xmm1, xmm2/m128, <XMM0>	RM0	V/V	SSE4_1	Select packed single precision floating-point values from xmm1 and xmm2/m128 from mask specified in XMM0 and store the values into xmm1.
VEX.128.66.0F3A.W0 4A /r /is4 VBLENDVPS xmm1, xmm2, xmm3/m128, xmm4	RVMR	V/V	AVX	Conditionally copy single precision floating-point values from xmm2 or xmm3/m128 to xmm1, based on mask bits in the specified mask operand, xmm4.
VEX.256.66.0F3A.W0 4A /r /is4 VBLENDVPS ymm1, ymm2, ymm3/m256, ymm4	RVMR	V/V	AVX	Conditionally copy single precision floating-point values from ymm2 or ymm3/m256 to ymm1, based on mask bits in the specified mask register, ymm4.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM0	ModRM:reg (r, w)	ModRM:r/m (r)	implicit XMM0	N/A
RVMR	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8[7:4]

Description

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

Each quadword element of the destination operand is copied from:

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

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

128-bit Legacy SSE version: The first source operand and the destination operand is the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute BLENDVPS with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand is an XMM register or 128-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (MAXVL-1:128) of the corresponding YMM register (destination register) are zeroed. VEX.W must be 0, otherwise, the instruction will #UD.

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

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

Operation

BLENDVPS (128-bit Legacy SSE Version)

```
MASK := XMM0
IF (MASK[31] = 0) THEN DEST[31:0] := DEST[31:0]
        ELSE DEST [31:0] := SRC[31:0] FI
IF (MASK[63] = 0) THEN DEST[63:32] := DEST[63:32]
        ELSE DEST [63:32] := SRC[63:32] FI
IF (MASK[95] = 0) THEN DEST[95:64] := DEST[95:64]
        ELSE DEST [95:64] := SRC[95:64] FI
IF (MASK[127] = 0) THEN DEST[127:96] := DEST[127:96]
        ELSE DEST [127:96] := SRC[127:96] FI
DEST[MAXVL-1:128] (Unmodified)
```

VBLENDVPS (VEX.128 Encoded Version)

```
MASK := SRC3
IF (MASK[31] = 0) THEN DEST[31:0] := SRC1[31:0]
        ELSE DEST [31:0] := SRC2[31:0] FI
IF (MASK[63] = 0) THEN DEST[63:32] := SRC1[63:32]
        ELSE DEST [63:32] := SRC2[63:32] FI
IF (MASK[95] = 0) THEN DEST[95:64] := SRC1[95:64]
        ELSE DEST [95:64] := SRC2[95:64] FI
IF (MASK[127] = 0) THEN DEST[127:96] := SRC1[127:96]
        ELSE DEST [127:96] := SRC2[127:96] FI
DEST[MAXVL-1:128] := 0
```

VBLENDVPS (VEX.256 Encoded Version)

```
MASK := SRC3
IF (MASK[31] = 0) THEN DEST[31:0] := SRC1[31:0]
        ELSE DEST [31:0] := SRC2[31:0] FI
IF (MASK[63] = 0) THEN DEST[63:32] := SRC1[63:32]
        ELSE DEST [63:32] := SRC2[63:32] FI
IF (MASK[95] = 0) THEN DEST[95:64] := SRC1[95:64]
        ELSE DEST [95:64] := SRC2[95:64] FI
IF (MASK[127] = 0) THEN DEST[127:96] := SRC1[127:96]
        ELSE DEST [127:96] := SRC2[127:96] FI
IF (MASK[159] = 0) THEN DEST[159:128] := SRC1[159:128]
        ELSE DEST [159:128] := SRC2[159:128] FI
IF (MASK[191] = 0) THEN DEST[191:160] := SRC1[191:160]
        ELSE DEST [191:160] := SRC2[191:160] FI
IF (MASK[223] = 0) THEN DEST[223:192] := SRC1[223:192]
        ELSE DEST [223:192] := SRC2[223:192] FI
IF (MASK[255] = 0) THEN DEST[255:224] := SRC1[255:224]
        ELSE DEST [255:224] := SRC2[255:224] FI
```

Intel C/C++ Compiler Intrinsic Equivalent

BLENDVPS __m128 _mm_blendv_ps(__m128 v1, __m128 v2, __m128 v3);
VBLENDVPS __m128 _mm_blendv_ps (__m128 a, __m128 b, __m128 mask);
VBLENDVPS __m256 _mm256_blendv_ps (__m256 a, __m256 b, __m256 mask);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions," additionally:

#UD	If VEX.W = 1.

BLSI—Extract Lowest Set Isolated Bit

Opcode/Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.LZ.0F38.W0 F3 /3 BLSI r32, r/m32	VM	V/V	BMI1	Extract lowest set bit from r/m32 and set that bit in r32.
VEX.LZ.0F38.W1 F3 /3 BLSI r64, r/m64	VM	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
VM	VEX.vvvv (w)	ModRM:r/m (r)	N/A	N/A

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 2-29, "Type 13 Class Exception Conditions."

BLSMSK—Get Mask Up to Lowest Set Bit

Opcode/Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.LZ.0F38.W0 F3 /2 BLSMSK r32, r/m32	VM	V/V	BMI1	Set all lower bits in r32 to "1" starting from bit 0 to lowest set bit in r/m32.
VEX.LZ.0F38.W1 F3 /2 BLSMSK r64, r/m64	VM	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
VM	VEX.vvvv (w)	ModRM:r/m (r)	N/A	N/A

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 2-29, "Type 13 Class Exception Conditions."

BLSR—Reset Lowest Set Bit

Opcode/Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.LZ.0F38.W0 F3 /1 BLSR r32, r/m32	VM	V/V	BMI1	Reset lowest set bit of r/m32, keep all other bits of r/m32 and write result to r32.
VEX.LZ.0F38.W1 F3 /1 BLSR r64, r/m64	VM	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
VM	VEX.vvvv (w)	ModRM:r/m (r)	N/A	N/A

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 2-29, "Type 13 Class Exception Conditions."

BNDCL—Check Lower Bound

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 1A /r BNDCL bnd, r/m32	RM	N.E./V	MPX	Generate a #BR if the address in r/m32 is lower than the lower bound in bnd.LB.
F3 0F 1A /r BNDCL bnd, r/m64	RM	V/N.E.	MPX	Generate a #BR if the address in r/m64 is lower than the lower bound in bnd.LB.

Instruction Operand Encoding

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

Description

Compare the address in the second operand with the lower bound in bnd. The second operand can be either a register or memory operand. If the address is lower than the lower bound in bnd.LB, it will set BNDSTATUS to 01H and signal a #BR exception.

This instruction does not cause any memory access, and does not read or write any flags.

Operation

BNDCL BND, reg

```
IF reg < BND.LB Then
    BNDSTATUS := 01H;
    #BR;
FI;
```

BNDCL BND, mem

```
TEMP := LEA(mem);
IF TEMP < BND.LB Then
    BNDSTATUS := 01H;
    #BR;
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

BNDCL void _bnd_chk_ptr_lbounds(const void *q)

Flags Affected

None

Protected Mode Exceptions

#BR	If lower bound check fails.
#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 67H prefix is not used and CS.D=0.
	If 67H prefix is used and CS.D=1.

Real-Address Mode Exceptions

#BR	If lower bound check fails.
#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.

Virtual-8086 Mode Exceptions

#BR	If lower bound check fails.
#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.

Same exceptions as in protected mode.

BNDCU/BNDCN—Check Upper Bound

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 1A /r BNDCU bnd, r/m32	RM	N.E./V	MPX	Generate a #BR if the address in r/m32 is higher than the upper bound in bnd.UB (bnb.UB in 1's complement form).
F2 0F 1A /r BNDCU bnd, r/m64	RM	V/N.E.	MPX	Generate a #BR if the address in r/m64 is higher than the upper bound in bnd.UB (bnb.UB in 1's complement form).
F2 0F 1B /r BNDCN bnd, r/m32	RM	N.E./V	MPX	Generate a #BR if the address in r/m32 is higher than the upper bound in bnd.UB (bnb.UB not in 1's complement form).
F2 0F 1B /r BNDCN bnd, r/m64	RM	V/N.E.	MPX	Generate a #BR if the address in r/m64 is higher than the upper bound in bnd.UB (bnb.UB not in 1's complement form).

Instruction Operand Encoding

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

Description

Compare the address in the second operand with the upper bound in bnd. The second operand can be either a register or a memory operand. If the address is higher than the upper bound in bnd.UB, it will set BNDSTATUS to 01H and signal a #BR exception.

BNDCU perform 1's complement operation on the upper bound of bnd first before proceeding with address comparison. BNDCN perform address comparison directly using the upper bound in bnd that is already reverted out of 1's complement form.

This instruction does not cause any memory access, and does not read or write any flags.

Effective address computation of m32/64 has identical behavior to LEA

Operation

BNDCU BND, reg

```
IF reg > NOT(BND.UB) Then
    BNDSTATUS := 01H;
    #BR;
FI;
```

BNDCU BND, mem

```
TEMP := LEA(mem);
IF TEMP > NOT(BND.UB) Then
    BNDSTATUS := 01H;
    #BR;
FI;
```

BNDCN BND, reg

```
IF reg > BND.UB Then
    BNDSTATUS := 01H;
    #BR;
FI;
```

BNDCN BND, mem

```
TEMP := LEA(mem);
IF TEMP > BND.UB Then
    BNDSTATUS := 01H;
    #BR;
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

BNDCU .void _bnd_chk_ptr_ubounds(const void *q)

Flags Affected

None

Protected Mode Exceptions

#BR	If upper bound check fails.
#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 67H prefix is not used and CS.D=0.
	If 67H prefix is used and CS.D=1.

Real-Address Mode Exceptions

#BR	If upper bound check fails.
#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.

Virtual-8086 Mode Exceptions

#BR	If upper bound check fails.
#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.

Same exceptions as in protected mode.

BNDLDX—Load Extended Bounds Using Address Translation

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 1A /r BNDLDX bnd, mib	RM	V/V	MPX	Load the bounds stored in a bound table entry (BTE) into bnd with address translation using the base of mib and conditional on the index of mib matching the pointer value in the BTE.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (w)	SIB.base (r): Address of pointer SIB.index(r)	N/A

Description

BNDLDX uses the linear address constructed from the base register and displacement of the SIB-addressing form of the memory operand (mib) to perform address translation to access a bound table entry and conditionally load the bounds in the BTE to the destination. The destination register is updated with the bounds in the BTE, if the content of the index register of mib matches the pointer value stored in the BTE.

If the pointer value comparison fails, the destination is updated with INIT bounds (lb = 0x0, ub = 0x0) (note: as articulated earlier, the upper bound is represented using 1's complement, therefore, the 0x0 value of upper bound allows for access to full memory).

This instruction does not cause memory access to the linear address of mib nor the effective address referenced by the base, and does not read or write any flags.

Segment overrides apply to the linear address computation with the base of mib, and are used during address translation to generate the address of the bound table entry. By default, the address of the BTE is assumed to be linear address. There are no segmentation checks performed on the base of mib.

The base of mib will not be checked for canonical address violation as it does not access memory.

Any encoding of this instruction that does not specify base or index register will treat those registers as zero (constant). The reg-reg form of this instruction will remain a NOP.

The scale field of the SIB byte has no effect on these instructions and is ignored.

The bound register may be partially updated on memory faults. The order in which memory operands are loaded is implementation specific.

Operation

```
base := mib.SIB.base ? mib.SIB.base + Disp: 0;
ptr_value := mib.SIB.index ? mib.SIB.index : 0;

Outside 64-bit Mode
A_BDE[31:0] := (Zero_extend32(base[31:12] « 2) + (BNDCFG[31:12] «12 );
A_BT[31:0] := LoadFrom(A_BDE );
IF A_BT[0] equal 0 Then
    BNDSTATUS := A_BDE | 02H;
    #BR;
FI;
A_BTE[31:0] := (Zero_extend32(base[11:2] « 4) + (A_BT[31:2] « 2 );
Temp_lb[31:0] := LoadFrom(A_BTE);
Temp_ub[31:0] := LoadFrom(A_BTE + 4);
Temp_ptr[31:0] := LoadFrom(A_BTE + 8);
IF Temp_ptr equal ptr_value Then
    BND.LB := Temp_lb;
    BND.UB := Temp_ub;
```

```
ELSE
    BND.LB := 0;
    BND.UB := 0;
FI;
```

In 64-bit Mode

```
A_BDE[63:0] := (Zero_extend64(base[47+MAWA:20] « 3) + (BNDCFG[63:12] «12 );[1]
A_BT[63:0] := LoadFrom(A_BDE);
IF A_BT[0] equal 0 Then
    BNDSTATUS := A_BDE | 02H;
    #BR;
FI;
A_BTE[63:0] := (Zero_extend64(base[19:3] « 5) + (A_BT[63:3] « 3 );
Temp_lb[63:0] := LoadFrom(A_BTE);
Temp_ub[63:0] := LoadFrom(A_BTE + 8);
Temp_ptr[63:0] := LoadFrom(A_BTE + 16);
IF Temp_ptr equal ptr_value Then
    BND.LB := Temp_lb;
    BND.UB := Temp_ub;
ELSE
    BND.LB := 0;
    BND.UB := 0;
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

BNDLDX: Generated by compiler as needed.

Flags Affected

None.

Protected Mode Exceptions

#BR	If the bound directory entry is invalid.
#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 67H prefix is not used and CS.D=0.
	If 67H prefix is used and CS.D=1.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.
	If DS register contains a NULL segment selector.
#PF(fault code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.

1. If CPL < 3, the supervisor MAWA (MAWAS) is used; this value is 0. If CPL = 3, the user MAWA (MAWAU) is used; this value is enumerated in CPUID.07H.00H:ECX.MPX_MAWAU[21:17]. See Appendix E.3.1 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.
#PF(fault code)	If a page fault occurs.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#BR	If the bound directory entry is invalid.
#UD	If ModRM is RIP relative.
	If the LOCK prefix is used.
	If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
#GP(0)	If the memory address (A_BDE or A_BTE) is in a non-canonical form.
#PF(fault code)	If a page fault occurs.

BNDMK—Make Bounds

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 1B /r BNDMK bnd, m32	RM	N.E./V	MPX	Make lower and upper bounds from m32 and store them in bnd.
F3 0F 1B /r BNDMK bnd, m64	RM	V/N.E.	MPX	Make lower and upper bounds from m64 and store them in bnd.

Instruction Operand Encoding

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

Description

Makes bounds from the second operand and stores the lower and upper bounds in the bound register bnd. The second operand must be a memory operand. The content of the base register from the memory operand is stored in the lower bound bnd.LB. The 1's complement of the effective address of m32/m64 is stored in the upper bound b.UB. Computation of m32/m64 has identical behavior to LEA.

This instruction does not cause any memory access, and does not read or write any flags.

If the instruction did not specify base register, the lower bound will be zero. The reg-reg form of this instruction retains legacy behavior (NOP).

The instruction causes an invalid-opcode exception (#UD) if executed in 64-bit mode with RIP-relative addressing.

Operation

```
BND.LB := SRCMEM.base;
IF 64-bit mode Then
    BND.UB := NOT(LEA.64_bits(SRCMEM));
ELSE
    BND.UB := Zero_Extend.64_bits(NOT(LEA.32_bits(SRCMEM)));
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

BNDMKvoid * _bnd_set_ptr_bounds(const void * q, size_t size);

Flags Affected

None.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 67H prefix is not used and CS.D=0.
	If 67H prefix is used and CS.D=1.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 16-bit addressing is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used. If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled. If RIP-relative addressing is used.
#SS(0)	If the memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.

Same exceptions as in protected mode.

BNDMOV—Move Bounds

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 1A /r BNDMOV bnd1, bnd2/m64	RM	N.E./V	MPX	Move lower and upper bound from bnd2/m64 to bound register bnd1.
66 0F 1A /r BNDMOV bnd1, bnd2/m128	RM	V/N.E.	MPX	Move lower and upper bound from bnd2/m128 to bound register bnd1.
66 0F 1B /r BNDMOV bnd1/m64, bnd2	MR	N.E./V	MPX	Move lower and upper bound from bnd2 to bnd1/m64.
66 0F 1B /r BNDMOV bnd1/m128, bnd2	MR	V/N.E.	MPX	Move lower and upper bound from bnd2 to bound register bnd1/m128.

Instruction Operand Encoding

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

Description

BNDMOV moves a pair of lower and upper bound values from the source operand (the second operand) to the destination (the first operand). Each operation is 128-bit move. The exceptions are same as the MOV instruction. The memory format for loading/store bounds in 64-bit mode is shown in Figure 3-5.

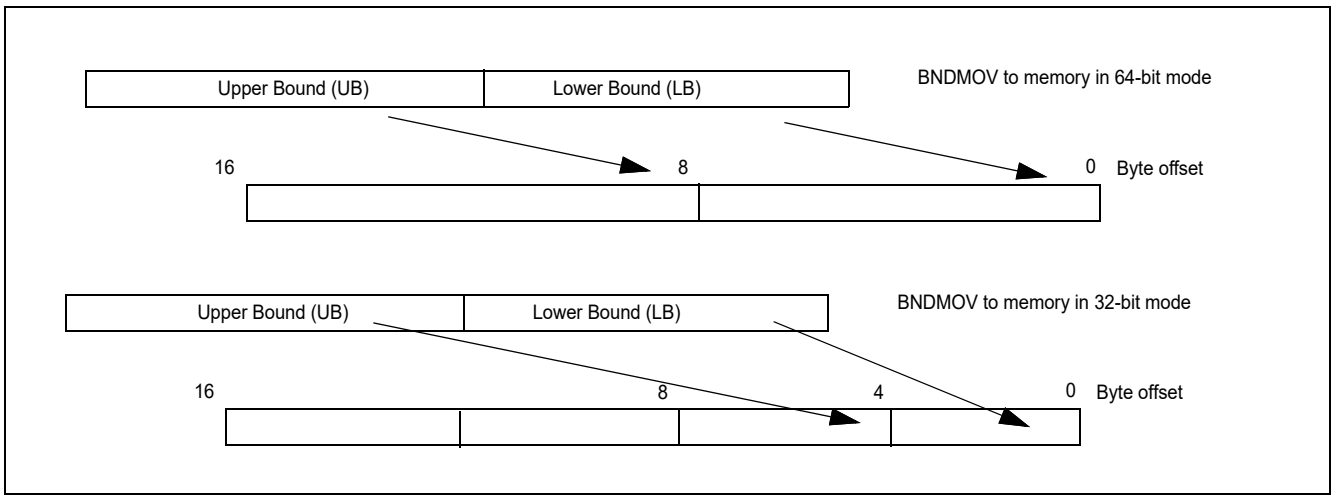


Figure 3-5. Memory Layout of BNDMOV to/from Memory

This instruction does not change flags.

Operation

BNDMOV register to register

```
DEST.LB := SRC.LB;
DEST.UB := SRC.UB;
```

BNDMOV from memory

```
IF 64-bit mode THEN
        DEST.LB := LOAD_QWORD(SRC);
        DEST.UB := LOAD_QWORD(SRC+8);
    ELSE
        DEST.LB := LOAD_DWORD_ZERO_EXT(SRC);
        DEST.UB := LOAD_DWORD_ZERO_EXT(SRC+4);
FI;
```

BNDMOV to memory

```
IF 64-bit mode THEN
        DEST[63:0] := SRC.LB;
        DEST[127:64] := SRC.UB;
    ELSE
        DEST[31:0] := SRC.LB;
        DEST[63:32] := SRC.UB;
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

BNDMOV void * _bnd_copy_ptr_bounds(const void *q, const void *r)

Flags Affected

None.

Protected Mode Exceptions

#UD	If the LOCK prefix is used but the destination is not a memory operand.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 67H prefix is not used and CS.D=0.
	If 67H prefix is used and CS.D=1.
#SS(0)	If the memory operand effective address is outside the SS segment limit.
#GP(0)	If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the destination operand points to a non-writable segment
	If the DS, ES, FS, or GS segment register contains a NULL segment selector.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL is 3.
#PF(fault code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used but the destination is not a memory operand.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.
#GP(0)	If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If the memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used but the destination is not a memory operand. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 16-bit addressing is used.
#GP(0)	If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If the memory operand effective address is outside the SS segment limit.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL is 3.
#PF(fault code)	If a page fault occurs.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used but the destination is not a memory operand. If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
#SS(0)	If the memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL is 3.
#PF(fault code)	If a page fault occurs.

BNDSTX—Store Extended Bounds Using Address Translation

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 1B /r BNDSTX mib, bnd	MR	V/V	MPX	Store the bounds in bnd and the pointer value in the index register of mib to a bound table entry (BTE) with address translation using the base of mib.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
MR	SIB.base (r): Address of pointer SIB.index(r)	ModRM:reg (r)	N/A

Description

BNDSTX uses the linear address constructed from the displacement and base register of the SIB-addressing form of the memory operand (mib) to perform address translation to store to a bound table entry. The bounds in the source operand bnd are written to the lower and upper bounds in the BTE. The content of the index register of mib is written to the pointer value field in the BTE.

This instruction does not cause memory access to the linear address of mib nor the effective address referenced by the base, and does not read or write any flags.

Segment overrides apply to the linear address computation with the base of mib, and are used during address translation to generate the address of the bound table entry. By default, the address of the BTE is assumed to be linear address. There are no segmentation checks performed on the base of mib.

The base of mib will not be checked for canonical address violation as it does not access memory.

Any encoding of this instruction that does not specify base or index register will treat those registers as zero (constant). The reg-reg form of this instruction will remain a NOP.

The scale field of the SIB byte has no effect on these instructions and is ignored.

The bound register may be partially updated on memory faults. The order in which memory operands are loaded is implementation specific.

Operation

```
base := mib.SIB.base ? mib.SIB.base + Disp: 0;
ptr_value := mib.SIB.index ? mib.SIB.index : 0;

Outside 64-bit Mode
A_BDE[31:0] := (Zero_extend32(base[31:12] « 2) + (BNDCFG[31:12] «12 );
A_BT[31:0] := LoadFrom(A_BDE);
IF A_BT[0] equal 0 Then
    BNDSTATUS := A_BDE | 02H;
    #BR;
FI;
A_DEST[31:0] := (Zero_extend32(base[11:2] « 4) + (A_BT[31:2] « 2 ); // address of Bound table entry
A_DEST[8][31:0] := ptr_value;
A_DEST[0][31:0] := BND.LB;
A_DEST[4][31:0] := BND.UB;
```

In 64-bit Mode

```
A_BDE[63:0] := (Zero_extend64(base[47+MAWA:20] « 3) + (BNDCFG[63:12] «12 );[1]
A_BT[63:0] := LoadFrom(A_BDE);
IF A_BT[0] equal 0 Then
    BNDSTATUS := A_BDE | 02H;
    #BR;
FI;
A_DEST[63:0] := (Zero_extend64(base[19:3] « 5) + (A_BT[63:3] « 3 ); // address of Bound table entry
A_DEST[16][63:0] := ptr_value;
A_DEST[0][63:0] := BND.LB;
A_DEST[8][63:0] := BND.UB;
```

Intel C/C++ Compiler Intrinsic Equivalent

BNDSTX: _bnd_store_ptr_bounds(const void **ptr_addr, const void *ptr_val);

Flags Affected

None.

Protected Mode Exceptions

#BR	If the bound directory entry is invalid.
#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 67H prefix is not used and CS.D=0.
	If 67H prefix is used and CS.D=1.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.
	If DS register contains a NULL segment selector.
	If the destination operand points to a non-writable segment
#PF(fault code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.
#PF(fault code)	If a page fault occurs.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

---

1. If CPL < 3, the supervisor MAWA (MAWAS) is used; this value is 0. If CPL = 3, the user MAWA (MAWAU) is used; this value is enumerated in CPUID.07H.00H:ECX.MPX_MAWAU[21:17]. See Appendix E.3.1 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

64-Bit Mode Exceptions

#BR	If the bound directory entry is invalid.
#UD	If ModRM is RIP relative.
	If the LOCK prefix is used.
	If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
#GP(0)	If the memory address (A_BDE or A_BTE) is in a non-canonical form.
	If the destination operand points to a non-writable segment
#PF(fault code)	If a page fault occurs.

BOUND—Check Array Index Against Bounds

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
62 /r	BOUND r16, m16&16	RM	Invalid	Valid	Check if r16 (array index) is within bounds specified by m16&16.
62 /r	BOUND r32, m32&32	RM	Invalid	Valid	Check if r32 (array index) is within bounds specified by m32&32.

Instruction Operand Encoding

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

Description

BOUND determines if the first operand (array index) is within the bounds of an array specified the second operand (bounds operand). The array index is a signed integer located in a register. The bounds operand is a memory location that contains a pair of signed doubleword-integers (when the operand-size attribute is 32) or a pair of signed word-integers (when the operand-size attribute is 16). The first doubleword (or word) is the lower bound of the array and the second doubleword (or word) is the upper bound of the array. The array index must be greater than or equal to the lower bound and less than or equal to the upper bound plus the operand size in bytes. If the index is not within bounds, a BOUND range exceeded exception (#BR) is signaled. When this exception is generated, the saved return instruction pointer points to the BOUND instruction.

The bounds limit data structure (two words or doublewords containing the lower and upper limits of the array) is usually placed just before the array itself, making the limits addressable via a constant offset from the beginning of the array. Because the address of the array already will be present in a register, this practice avoids extra bus cycles to obtain the effective address of the array bounds.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

```
IF 64bit Mode
    THEN
        #UD;
    ELSE
        IF (ArrayIndex < LowerBound OR ArrayIndex > UpperBound) THEN
        (* Below lower bound or above upper bound *)
            IF <equation for PL enabled> THEN BNDSTATUS := 0
            #BR;
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#BR	If the bounds test fails.
#UD	If second operand is not a memory location.
	If the LOCK prefix is used.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#BR	If the bounds test fails.
#UD	If second operand is not a memory location.
	If the LOCK prefix is used.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#BR	If the bounds test fails.
#UD	If second operand is not a memory location.
	If the LOCK prefix is used.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If in 64-bit mode.

BSF—Bit Scan Forward

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
0F BC /r	BSF r16, r/m16	RM	Valid	Valid	Bit scan forward on r/m16.
0F BC /r	BSF r32, r/m32	RM	Valid	Valid	Bit scan forward on r/m32.
REX.W + 0F BC /r	BSF r64, r/m64	RM	Valid	N.E.	Bit scan forward on r/m64.

Instruction Operand Encoding

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

Description

Searches the source operand (second operand) for the least significant set bit (1 bit). If a least significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the content of the source operand is zero, the destination operand is unmodified.[1]

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

Operation

```
IF SRC <> 0
    THEN
        temp := 0;
        WHILE Bit(SRC, temp) = 0
        DO
            temp := temp + 1;
        OD;
        DEST := temp;
FI;
```

Flags Affected

The ZF flag is set to 1 if the source operand is 0; otherwise, the ZF flag is cleared. The PF flag is set to 1 if the number of bits set in the source operand is even; otherwise, it is cleared. The CF, OF, SF, and AF flags are all cleared.[2]

Protected Mode Exceptions

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

1. With a zero source operand on some older processors, use of a 32-bit operand size may clear the upper 32 bits of a 64-bit destination while leaving the lower 32 bits unmodified.
2. On some older processors, the CF, OF, SF, AF, and PF flags are unmodified.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

BSR—Bit Scan Reverse

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
0F BD /r	BSR r16, r/m16	RM	Valid	Valid	Bit scan reverse on r/m16.
0F BD /r	BSR r32, r/m32	RM	Valid	Valid	Bit scan reverse on r/m32.
REX.W + 0F BD /r	BSR r64, r/m64	RM	Valid	N.E.	Bit scan reverse on r/m64.

Instruction Operand Encoding

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

Description

Searches the source operand (second operand) for the most significant set bit (1 bit). If a most significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the content source operand is zero, the destination operand is unmodified.[1]

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

Operation

```
IF SRC <> 0
        temp := OperandSize – 1;
        WHILE Bit(SRC, temp) = 0
        DO
            temp := temp - 1;
        OD;
        DEST := temp;
FI;
```

Flags Affected

The ZF flag is set to 1 if the source operand is 0; otherwise, the ZF flag is cleared. The PF flag is set to 1 if the number of bits set in the source operand is even; otherwise, it is cleared. The CF, OF, SF, and AF flags are all cleared.[2]

Protected Mode Exceptions

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

1. With a zero source operand on some older processors, use of a 32-bit operand size may clear the upper 32 bits of a 64-bit destination while leaving the lower 32 bits unmodified.
2. On some older processors, the CF, OF, SF, AF, and PF flags are unmodified.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

BSWAP—Byte Swap

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
0F C8+*rd*	BSWAP *r32*	O	Valid*	Valid	Reverses the byte order of a 32-bit register.
REX.W + 0F C8+*rd*	BSWAP *r64*	O	Valid	N.E.	Reverses the byte order of a 64-bit register.

NOTES:

* See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
O	opcode + rd (r, w)	N/A	N/A	N/A

Description

Reverses the byte order of a 32-bit or 64-bit (destination) register. This instruction is provided for converting little-endian values to big-endian format and vice versa. To swap bytes in a word value (16-bit register), use the XCHG instruction. When the BSWAP instruction references a 16-bit register, the result is undefined.

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 bits. See the summary chart at the beginning of this section for encoding data and limits.

IA-32 Architecture Legacy Compatibility

The BSWAP instruction is not supported on IA-32 processors earlier than the Intel486™ processor family. For compatibility with this instruction, software should include functionally equivalent code for execution on Intel processors earlier than the Intel486 processor family.

Operation

```
TEMP := DEST
IF 64-bit mode AND OperandSize = 64
    THEN
        DEST[7:0] := TEMP[63:56];
        DEST[15:8] := TEMP[55:48];
        DEST[23:16] := TEMP[47:40];
        DEST[31:24] := TEMP[39:32];
        DEST[39:32] := TEMP[31:24];
        DEST[47:40] := TEMP[23:16];
        DEST[55:48] := TEMP[15:8];
        DEST[63:56] := TEMP[7:0];
    ELSE
        DEST[7:0] := TEMP[31:24];
        DEST[15:8] := TEMP[23:16];
        DEST[23:16] := TEMP[15:8];
        DEST[31:24] := TEMP[7:0];
FI;
```

Flags Affected

None.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

BT—Bit Test

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
0F A3 /r	BT r/m16, r16	MR	Valid	Valid	Store selected bit in CF flag.
0F A3 /r	BT r/m32, r32	MR	Valid	Valid	Store selected bit in CF flag.
REX.W + 0F A3 /r	BT r/m64, r64	MR	Valid	N.E.	Store selected bit in CF flag.
0F BA /4 ib	BT r/m16, imm8	MI	Valid	Valid	Store selected bit in CF flag.
0F BA /4 ib	BT r/m32, imm8	MI	Valid	Valid	Store selected bit in CF flag.
REX.W + 0F BA /4 ib	BT r/m64, imm8	MI	Valid	N.E.	Store selected bit in CF flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r)	ModRM:reg (r)	N/A	N/A
MI	ModRM:r/m (r)	imm8	N/A	N/A

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset (specified by the second operand) and stores the value of the bit in the CF flag. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode).
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-11.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. In this case, the low-order 3 or 5 bits (3 for 16-bit operands, 5 for 32-bit operands) of the immediate bit offset are stored in the immediate bit offset field, and the high-order bits are shifted and combined with the byte displacement in the addressing mode by the assembler. The processor will ignore the high order bits if they are not zero.

When accessing a bit in memory, the processor may access 4 bytes starting from the memory address for a 32-bit operand size, using by the following relationship:

Effective Address + (4 * (BitOffset DIV 32))

Or, it may access 2 bytes starting from the memory address for a 16-bit operand, using this relationship:

Effective Address + (2 * (BitOffset DIV 16))

It may do so even when only a single byte needs to be accessed to reach the given bit. When using this bit addressing mechanism, software should avoid referencing areas of memory close to address space holes. In particular, it should avoid references to memory-mapped I/O registers. Instead, software should use the MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

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

Operation

```
CF := Bit(BitBase, BitOffset);
```

Flags Affected

The CF flag contains the value of the selected bit. The ZF flag is unaffected. The OF, SF, AF, and PF flags are undefined.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

BTC—Bit Test and Complement

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
0F BB /r	BTC r/m16, r16	MR	Valid	Valid	Store selected bit in CF flag and complement.
0F BB /r	BTC r/m32, r32	MR	Valid	Valid	Store selected bit in CF flag and complement.
REX.W + 0F BB /r	BTC r/m64, r64	MR	Valid	N.E.	Store selected bit in CF flag and complement.
0F BA /7 ib	BTC r/m16, imm8	MI	Valid	Valid	Store selected bit in CF flag and complement.
0F BA /7 ib	BTC r/m32, imm8	MI	Valid	Valid	Store selected bit in CF flag and complement.
REX.W + 0F BA /7 ib	BTC r/m64, imm8	MI	Valid	N.E.	Store selected bit in CF flag and complement.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r)	N/A	N/A
MI	ModRM:r/m (r, w)	imm8	N/A	N/A

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and complements the selected bit in the bit string. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode). This allows any bit position to be selected.
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-11.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See "BT—Bit Test" in this chapter for more information on this addressing mechanism.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.B permits access to additional registers (R8-R15) for the bit base. Using a REX prefix in the form of REX.R permits access to R8-R15 for the bit offset (when it uses a register). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
CF := Bit(BitBase, BitOffset);
Bit(BitBase, BitOffset) := NOT Bit(BitBase, BitOffset);
```

Flags Affected

The CF flag contains the value of the selected bit before it is complemented. The ZF flag is unaffected. The OF, SF, AF, and PF flags are undefined.

Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

BTR—Bit Test and Reset

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
0F B3 /r	BTR r/m16, r16	MR	Valid	Valid	Store selected bit in CF flag and clear.
0F B3 /r	BTR r/m32, r32	MR	Valid	Valid	Store selected bit in CF flag and clear.
REX.W + 0F B3 /r	BTR r/m64, r64	MR	Valid	N.E.	Store selected bit in CF flag and clear.
0F BA /6 ib	BTR r/m16, imm8	MI	Valid	Valid	Store selected bit in CF flag and clear.
0F BA /6 ib	BTR r/m32, imm8	MI	Valid	Valid	Store selected bit in CF flag and clear.
REX.W + 0F BA /6 ib	BTR r/m64, imm8	MI	Valid	N.E.	Store selected bit in CF flag and clear.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r)	N/A	N/A
MI	ModRM:r/m (r, w)	imm8	N/A	N/A

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and clears the selected bit in the bit string to 0. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode). This allows any bit position to be selected.
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-11.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See "BT—Bit Test" in this chapter for more information on this addressing mechanism.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

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

Operation

```
CF := Bit(BitBase, BitOffset);
Bit(BitBase, BitOffset) := 0;
```

Flags Affected

The CF flag contains the value of the selected bit before it is cleared. The ZF flag is unaffected. The OF, SF, AF, and PF flags are undefined.

Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

BTS—Bit Test and Set

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
0F AB /r	BTS r/m16, r16	MR	Valid	Valid	Store selected bit in CF flag and set.
0F AB /r	BTS r/m32, r32	MR	Valid	Valid	Store selected bit in CF flag and set.
REX.W + 0F AB /r	BTS r/m64, r64	MR	Valid	N.E.	Store selected bit in CF flag and set.
0F BA /5 ib	BTS r/m16, imm8	MI	Valid	Valid	Store selected bit in CF flag and set.
0F BA /5 ib	BTS r/m32, imm8	MI	Valid	Valid	Store selected bit in CF flag and set.
REX.W + 0F BA /5 ib	BTS r/m64, imm8	MI	Valid	N.E.	Store selected bit in CF flag and set.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r)	N/A	N/A
MI	ModRM:r/m (r, w)	imm8	N/A	N/A

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and sets the selected bit in the bit string to 1. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode). This allows any bit position to be selected.
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-11.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See "BT—Bit Test" in this chapter for more information on this addressing mechanism.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

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

Operation

```
CF := Bit(BitBase, BitOffset);
Bit(BitBase, BitOffset) := 1;
```

Flags Affected

The CF flag contains the value of the selected bit before it is set. The ZF flag is unaffected. The OF, SF, AF, and PF flags are undefined.

Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

BZHI—Zero High Bits Starting with Specified Bit Position

Opcode/Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.LZ.0F38.W0 F5 /r BZHI r32a, r/m32, r32b	RMV	V/V	BMI2	Zero bits in r/m32 starting with the position in r32b, write result to r32a.
VEX.LZ.0F38.W1 F5 /r BZHI r64a, r/m64, r64b	RMV	V/N.E.	BMI2	Zero bits in r/m64 starting with the position in r64b, write result to r64a.

Instruction Operand Encoding

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

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 and SF flags are updated based on the result. CF flag is set as specified in the Operation section. 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 2-29, "Type 13 Class Exception Conditions."

CALL—Call Procedure

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
E8 cw	CALL rel16	D	N.S.	Valid	Call near, relative, displacement relative to next instruction.
E8 cd	CALL rel32	D	Valid	Valid	Call near, relative, displacement relative to next instruction. 32-bit displacement sign extended to 64-bits in 64-bit mode.
FF /2	CALL r/m16	M	N.E.	Valid	Call near, absolute indirect, address given in r/m16.
FF /2	CALL r/m32	M	N.E.	Valid	Call near, absolute indirect, address given in r/m32.
FF /2	CALL r/m64	M	Valid	N.E.	Call near, absolute indirect, address given in r/m64.
9A cd	CALL ptr16:16	D	Invalid	Valid	Call far, absolute, address given in operand.
9A cp	CALL ptr16:32	D	Invalid	Valid	Call far, absolute, address given in operand.
FF /3	CALL m16:16	M	Valid	Valid	Call far, absolute indirect address given in m16:16. In 32-bit mode: if selector points to a gate, then RIP = 32-bit zero extended displacement taken from gate; else RIP = zero extended 16-bit offset from far pointer referenced in the instruction.
FF /3	CALL m16:32	M	Valid	Valid	In 64-bit mode: If selector points to a gate, then RIP = 64-bit displacement taken from gate; else RIP = zero extended 32-bit offset from far pointer referenced in the instruction.
REX.W FF /3	CALL m16:64	M	Valid	N.E.	In 64-bit mode: If selector points to a gate, then RIP = 64-bit displacement taken from gate; else RIP = 64-bit offset from far pointer referenced in the instruction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
D	Offset	N/A	N/A	N/A
M	ModRM:r/m (r)	N/A	N/A	N/A

Description

Saves procedure linking information on the stack and branches to the called procedure specified using the target operand. The target operand specifies the address of the first instruction in the called procedure. The operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four types of calls:

- **Near Call** — A call to a procedure in the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intra-segment call.
- **Far Call** — A call to a procedure located in a different segment than the current code segment, sometimes referred to as an inter-segment call.
- **Inter-privilege-level far call** — A far call to a procedure in a segment at a different privilege level than that of the currently executing program or procedure.
- **Task switch** — A call to a procedure located in a different task.

The latter two call types (inter-privilege-level call and task switch) can only be executed in protected mode. See "Calling Procedures Using Call and RET" in Chapter 6 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for additional information on near, far, and inter-privilege-level calls. See Chapter 10, "Task Management," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, for information on performing task switches with the CALL instruction.

Near Call. When executing a near call, the processor pushes the value of the EIP register (which contains the offset of the instruction following the CALL instruction) on the stack (for use later as a return-instruction pointer). The processor then branches to the address in the current code segment specified by the target operand. The target operand specifies either an absolute offset in the code segment (an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current value of the instruction pointer in the EIP register; this value points to the instruction following the CALL instruction). The CS register is not changed on near calls.

For a near call absolute, an absolute offset is specified indirectly in a general-purpose register or a memory location (*r/m16*, *r/m32, or r/m64*). The operand-size attribute determines the size of the target operand (16, 32 or 64 bits). When in 64-bit mode, the operand size for near call (and all near branches) is forced to 64-bits. Absolute offsets are loaded directly into the EIP(RIP) register. If the operand size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits. When accessing an absolute offset indirectly using the stack pointer [ESP] as the base register, the base value used is the value of the ESP before the instruction executes.

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(RIP) register. In 64-bit mode the relative offset is always a 32-bit immediate value which is sign extended to 64-bits before it is added to the value in the RIP register for the target calculation. As with absolute offsets, the operand-size attribute determines the size of the target operand (16, 32, or 64 bits). In 64-bit mode the target operand will always be 64-bits because the operand size is forced to 64-bits for near branches.

Far Calls in Real-Address or Virtual-8086 Mode. When executing a far call in real- address or virtual-8086 mode, the processor pushes the current value of both the CS and EIP registers on the stack for use as a return-instruction pointer. The processor then performs a "far branch" to the code segment and offset specified with the target operand for the called procedure. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). With the pointer method, the segment and offset of the called procedure is encoded in the instruction using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared.

Far Calls in Protected Mode. When the processor is operating in protected mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level
- Far call to a different privilege level (inter-privilege level call)
- Task switch (far call to another task)

In protected mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate, task gate, or TSS) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in protected mode is very similar to one carried out in real-address or virtual-8086 mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The operand- size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register; the offset from the instruction is loaded into the EIP register.

A call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. Using this mechanism provides an extra level of indirection and is the preferred method of making calls between 16-bit and 32-bit code segments.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a call gate. The segment selector specified by the target operand identifies the call gate. The target operand can specify the call gate segment selector either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch. (Note that when using a call gate to perform a far call to a segment at the same privilege level, no stack switch occurs.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack, an optional set of parameters from the calling procedures stack, and the segment selector and instruction pointer for the calling procedure's code segment. (A value in the call gate descriptor determines how many parameters to copy to the new stack.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Executing a task switch with the CALL instruction is similar to executing a call through a call gate. The target operand specifies the segment selector of the task gate for the new task activated by the switch (the offset in the target operand is ignored). The task gate in turn points to the TSS for the new task, which contains the segment selectors for the task's code and stack segments. Note that the TSS also contains the EIP value for the next instruction that was to be executed before the calling task was suspended. This instruction pointer value is loaded into the EIP register to re-start the calling task.

The CALL instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 10, "Task Management," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, for information on the mechanics of a task switch.

When execution of a CALL instruction effects a task switch, the nested task flag (NT) is set in the EFLAGS register and the new TSS's previous task link field is loaded with the old task's TSS selector. Code is expected to suspend this nested task by executing an IRET instruction which, because the NT flag is set, automatically uses the previous task link to return to the calling task. (See "Task Linking" in Chapter 10 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, for information on nested tasks.) Switching tasks with the CALL instruction differs in this regard from JMP instruction. JMP does not set the NT flag and therefore does not expect an IRET instruction to suspend the task.

Mixing 16-Bit and 32-Bit Calls. When making far calls between 16-bit and 32-bit code segments, use a call gate. If the far call is from a 32-bit code segment to a 16-bit code segment, the call should be made from the first 64 KBytes of the 32-bit code segment. This is because the operand-size attribute of the instruction is set to 16, so only a 16-bit return address offset can be saved. Also, the call should be made using a 16-bit call gate so that 16-bit values can be pushed on the stack. See Chapter 24, "Mixing 16-Bit and 32-Bit Code," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B, for more information.

Far Calls in Compatibility Mode. When the processor is operating in compatibility mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level, remaining in compatibility mode
- Far call to the same privilege level, transitioning to 64-bit mode
- Far call to a different privilege level (inter-privilege level call), transitioning to 64-bit mode

Note that a CALL instruction cannot be used to cause a task switch in compatibility mode since task switches are not supported in IA-32e mode.

In compatibility mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in compatibility mode is very similar to one carried out in protected mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register and the offset from the instruction is loaded into the EIP register. The difference is that 64-bit mode may be entered. This specified by the L bit in the new code segment descriptor.

Note that a 64-bit call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. However, using this mechanism requires that the target code segment descriptor have the L bit set, causing an entry to 64-bit mode.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a 64-bit call gate. The segment selector specified by the target operand identifies the call gate. The target

operand can specify the call gate segment selector either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the 16-byte call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is set to NULL. The new stack pointer is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch. (Note that when using a call gate to perform a far call to a segment at the same privilege level, an implicit stack switch occurs as a result of entering 64-bit mode. The SS selector is unchanged, but stack segment accesses use a segment base of 0x0, the limit is ignored, and the default stack size is 64-bits. The full value of RSP is used for the offset, of which the upper 32-bits are undefined.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack and the segment selector and instruction pointer for the calling procedure's code segment. (Parameter copy is not supported in IA-32e mode.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Near/(Far) Calls in 64-bit Mode. When the processor is operating in 64-bit mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level, transitioning to compatibility mode
- Far call to the same privilege level, remaining in 64-bit mode
- Far call to a different privilege level (inter-privilege level call), remaining in 64-bit mode

Note that in this mode the CALL instruction can not be used to cause a task switch in 64-bit mode since task switches are not supported in IA-32e mode.

In 64-bit mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in 64-bit mode is very similar to one carried out in compatibility mode. The target operand specifies an absolute far address indirectly with a memory location (*m16:16, m16:32* or *m16:64*). The form of CALL with a direct specification of absolute far address is not defined in 64-bit mode. The operand-size attribute determines the size of the offset (16, 32, or 64 bits) in the far address. The new code segment selector and its descriptor are loaded into the CS register; the offset from the instruction is loaded into the EIP register. The new code segment may specify entry either into compatibility or 64-bit mode, based on the L bit value.

A 64-bit call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. However, using this mechanism requires that the target code segment descriptor have the L bit set.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a 64-bit call gate. The segment selector specified by the target operand identifies the call gate. The target operand can only specify the call gate segment selector indirectly with a memory location (*m16:16, m16:32* or *m16:64*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the 16-byte call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is set to NULL. The new stack pointer is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch.

Note that when using a call gate to perform a far call to a segment at the same privilege level, an implicit stack switch occurs as a result of entering 64-bit mode. The SS selector is unchanged, but stack segment accesses use a segment base of 0x0, the limit is ignored, and the default stack size is 64-bits. (The full value of RSP is used for the offset.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack and the segment selector and instruction pointer for the calling procedure's code segment. (Parameter copy is not supported in IA-32e mode.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Refer to Chapter 6, "Procedure Calls, Interrupts, and Exceptions," and Chapter 18, "Control-flow Enforcement Technology (CET)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for CET details.

When FRED transitions are enabled, an execution of far CALL that references a call gate causes a general-protection exception, as does an execution of far CALL that would enter compatibility mode when CPL is 0.

Instruction ordering. Instructions following a far call may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the far call have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

Instructions sequentially following a near indirect CALL instruction (i.e., those not at the target) may be executed speculatively. If software needs to prevent this (e.g., in order to prevent a speculative execution side channel), then an LFENCE instruction opcode can be placed after the near indirect CALL in order to block speculative execution.

Operation

```
IF near call
    THEN IF near relative call
        THEN
            IF OperandSize = 64
                THEN
                    tempDEST := SignExtend(DEST); (* DEST is rel32 *)
                    tempRIP := RIP + tempDEST;
                    IF stack not large enough for a 8-byte return address
                        THEN #SS(0); FI;
                    Push(RIP);
                    IF ShadowStackEnabled(CPL) AND DEST != 0
                        ShadowStackPush8B(RIP);
                    FI;
                    RIP := tempRIP;
            FI;
            IF OperandSize = 32
                THEN
                    tempEIP := EIP + DEST; (* DEST is rel32 *)
                    IF tempEIP is not within code segment limit THEN #GP(0); FI;
                    IF stack not large enough for a 4-byte return address
                        THEN #SS(0); FI;
                    Push(EIP);
                    IF ShadowStackEnabled(CPL) AND DEST != 0
                        ShadowStackPush4B(EIP);
                    FI;
                    EIP := tempEIP;
            FI;
            IF OperandSize = 16
                THEN
                    tempEIP := (EIP + DEST) AND 0000FFFFH; (* DEST is rel16 *)
                    IF tempEIP is not within code segment limit THEN #GP(0); FI;
                    IF stack not large enough for a 2-byte return address
                        THEN #SS(0); FI;
                    Push(IP);
                    IF ShadowStackEnabled(CPL) AND DEST != 0
                        (* IP is zero extended and pushed as a 32 bit value on shadow stack *)
                        ShadowStackPush4B(IP);
                    FI;
```

```
                EIP := tempEIP;
            FI;
        ELSE (* Near absolute call *)
            IF OperandSize = 64
                THEN
                    tempRIP := DEST; (* DEST is r/m64 *)
                    IF stack not large enough for a 8-byte return address
                        THEN #SS(0); FI;
                    Push(RIP);
                    IF ShadowStackEnabled(CPL)
                        ShadowStackPush8B(RIP);
                    FI;
                    RIP := tempRIP;
            FI;
            IF OperandSize = 32
                THEN
                    tempEIP := DEST; (* DEST is r/m32 *)
                    IF tempEIP is not within code segment limit THEN #GP(0); FI;
                    IF stack not large enough for a 4-byte return address
                        THEN #SS(0); FI;
                    Push(EIP);
                    IF ShadowStackEnabled(CPL)
                        ShadowStackPush4B(EIP);
                    FI;
                    EIP := tempEIP;
            FI;
            IF OperandSize = 16
                THEN
                    tempEIP := DEST AND 0000FFFFH; (* DEST is r/m16 *)
                    IF tempEIP is not within code segment limit THEN #GP(0); FI;
                    IF stack not large enough for a 2-byte return address
                        THEN #SS(0); FI;
                    Push(IP);
                    IF ShadowStackEnabled(CPL)
                        (* IP is zero extended and pushed as a 32 bit value on shadow stack *)
                        ShadowStackPush4B(IP);
                    FI;
                    EIP := tempEIP;
            FI;
    FI;rel/abs
    IF (Call near indirect, absolute indirect)
        IF EndbranchEnabledAndNotSuppressed(CPL)
            IF CPL = 3
                THEN
                    IF ( no 3EH prefix OR IA32_U_CET.NO_TRACK_EN == 0 )
                        THEN
                            IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
                    FI;
                ELSE
                    IF ( no 3EH prefix OR IA32_S_CET.NO_TRACK_EN == 0 )
                        THEN
                            IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                    FI;
            FI;
```

```
        FI;
    FI;
FI; near

IF far call and (PE = 0 or (PE = 1 and VM = 1)) (* Real-address or virtual-8086 mode *)
    THEN
        IF OperandSize = 32
            THEN
                IF stack not large enough for a 6-byte return address
                    THEN #SS(0); FI;
                IF DEST[31:16] is not zero THEN #GP(0); FI;
                Push(CS); (* Padded with 16 high-order bits *)
                Push(EIP);
                CS := DEST[47:32]; (* DEST is ptr16:32 or [m16:32] *)
                EIP := DEST[31:0]; (* DEST is ptr16:32 or [m16:32] *)
            ELSE (* OperandSize = 16 *)
                IF stack not large enough for a 4-byte return address
                    THEN #SS(0); FI;
                Push(CS);
                Push(IP);
                CS := DEST[31:16]; (* DEST is ptr16:16 or [m16:16] *)
                EIP := DEST[15:0]; (* DEST is ptr16:16 or [m16:16]; clear upper 16 bits *)
        FI;
FI;

IF far call and (PE = 1 and VM = 0) (* Protected mode or IA-32e Mode, not virtual-8086 mode*)
    THEN
        IF segment selector in target operand NULL
            THEN #GP(0); FI;
        IF segment selector index not within descriptor table limits
            THEN #GP(new code segment selector); FI;
        Read type and access rights of selected segment descriptor;
        IF IA32_EFER.LMA = 0
            THEN
                IF segment type is not a conforming or nonconforming code segment, call gate, task gate, or TSS
                    THEN #GP(segment selector); FI;
            ELSE
                IF segment type is not a conforming or nonconforming code segment or 64-bit call gate
                    THEN #GP(segment selector); FI;
        FI;
        Depending on type and access rights:
            GO TO CONFORMING-CODE-SEGMENT;
            GO TO NONCONFORMING-CODE-SEGMENT;
            GO TO CALL-GATE;
            GO TO TASK-GATE;
            GO TO TASK-STATE-SEGMENT;
FI;

CONFORMING-CODE-SEGMENT:
    IF L bit = 1 and D bit = 1 and IA32_EFER.LMA = 1
        THEN GP(new code segment selector); FI;
    IF DPL > CPL
        THEN #GP(new code segment selector); FI;
    IF CR4.FRED = 1 and CPL = 0 and L bit = 0
```

```
        THEN GP(new code segment selector); FI;
    IF segment not present
        THEN #NP(new code segment selector); FI;
    IF stack not large enough for return address
        THEN #SS(0); FI;
    tempEIP := DEST(Offset);
    IF target mode = Compatibility mode
        THEN tempEIP := tempEIP AND 00000000_FFFFFFFFH; FI;
    IF OperandSize = 16
        THEN
            tempEIP := tempEIP AND 0000FFFFH; FI; (* Clear upper 16 bits *)
    IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code segment limit)
        THEN #GP(0); FI;
    IF tempEIP is non-canonical
        THEN #GP(0); FI;
    IF ShadowStackEnabled(CPL)
        IF OperandSize = 32
            THEN
                tempPushLIP = CSBASE + EIP;
            ELSE
                IF OperandSize = 16
                    THEN
                        tempPushLIP = CSBASE + IP;
                    ELSE (* OperandSize = 64 *)
                        tempPushLIP = RIP;
                FI;
        FI;
        tempPushCS = CS;
    FI;
    IF OperandSize = 32
        THEN
            Push(CS); (* Padded with 16 high-order bits *)
            Push(EIP);
            CS := DEST(CodeSegmentSelector);
            (* Segment descriptor information also loaded *)
            CS(RPL) := CPL;
            EIP := tempEIP;
        ELSE
            IF OperandSize = 16
                THEN
                    Push(CS);
                    Push(IP);
                    CS := DEST(CodeSegmentSelector);
                    (* Segment descriptor information also loaded *)
                    CS(RPL) := CPL;
                    EIP := tempEIP;
                ELSE (* OperandSize = 64 *)
                    Push(CS); (* Padded with 48 high-order bits *)
                    Push(RIP);
                    CS := DEST(CodeSegmentSelector);
                    (* Segment descriptor information also loaded *)
                    CS(RPL) := CPL;
                    RIP := tempEIP;
            FI;
```

```
    FI;
    IF ShadowStackEnabled(CPL)
        IF (IA32_EFER.LMA and DEST(CodeSegmentSelector).L) = 0
            (* If target is legacy or compatibility mode then the SSP must be in low 4GB *)
            IF (SSP & 0xFFFFFFFF00000000 != 0)
                THEN #GP(0); FI;
        FI;
        (* align to 8 byte boundary if not already aligned *)
        tempSSP = SSP;
        Shadow_stack_store 4 bytes of 0 to (SSP – 4)
        SSP = SSP & 0xFFFFFFFFFFFFFFF8H
        ShadowStackPush8B(tempPushCS); (* Padded with 48 high-order bits of 0 *)
        ShadowStackPush8B(tempPushLIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
        ShadowStackPush8B(tempSSP);
    FI;
    IF EndbranchEnabled(CPL)
        IF CPL = 3
            THEN
                IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
                IA32_U_CET.SUPPRESS = 0
            ELSE
                IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                IA32_S_CET.SUPPRESS = 0
        FI;
    FI;
END;

NONCONFORMING-CODE-SEGMENT:
    IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
        THEN GP(new code segment selector); FI;
    IF (RPL > CPL) or (DPL ≠ CPL)
        THEN #GP(new code segment selector); FI;
    IF CR4.FRED = 1 and CPL = 0 and L bit = 0
        THEN GP(new code segment selector); FI;
    IF segment not present
        THEN #NP(new code segment selector); FI;
    IF stack not large enough for return address
        THEN #SS(0); FI;
    tempEIP := DEST(Offset);
    IF target mode = Compatibility mode
        THEN tempEIP := tempEIP AND 00000000_FFFFFFFFH; FI;
    IF OperandSize = 16
        THEN tempEIP := tempEIP AND 0000FFFFH; FI; (* Clear upper 16 bits *)
    IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code segment limit)
        THEN #GP(0); FI;
    IF tempEIP is non-canonical
        THEN #GP(0); FI;
    IF ShadowStackEnabled(CPL)
        IF IA32_EFER.LMA & CS.L
            tempPushLIP = RIP
        ELSE
            tempPushLIP = CSBASE + EIP;
        FI;
        tempPushCS = CS;
```

```
    FI;
    IF OperandSize = 32
        THEN
            Push(CS); (* Padded with 16 high-order bits *)
            Push(EIP);
            CS := DEST(CodeSegmentSelector);
            (* Segment descriptor information also loaded *)
            CS(RPL) := CPL;
            EIP := tempEIP;
        ELSE
            IF OperandSize = 16
                THEN
                    Push(CS);
                    Push(IP);
                    CS := DEST(CodeSegmentSelector);
                    (* Segment descriptor information also loaded *)
                    CS(RPL) := CPL;
                    EIP := tempEIP;
                ELSE (* OperandSize = 64 *)
                    Push(CS); (* Padded with 48 high-order bits *)
                    Push(RIP);
                    CS := DEST(CodeSegmentSelector);
                    (* Segment descriptor information also loaded *)
                    CS(RPL) := CPL;
                    RIP := tempEIP;
            FI;
    FI;
    IF ShadowStackEnabled(CPL)
        IF (IA32_EFER.LMA and DEST(CodeSegmentSelector).L) = 0
            (* If target is legacy or compatibility mode then the SSP must be in low 4GB *)
            IF (SSP & 0xFFFFFFFF00000000 != 0)
                THEN #GP(0); FI;
        FI;
    (* align to 8 byte boundary if not already aligned *)
    tempSSP = SSP;
    Shadow_stack_store 4 bytes of 0 to (SSP – 4)
    SSP = SSP & 0xFFFFFFFFFFFFFFF8H
    ShadowStackPush8B(tempPushCS); (* Padded with 48 high-order 0 bits *)
    ShadowStackPush8B(tempPushLIP); (* Padded 32 high-order bits of 0 for 32 bit LIP*)
    ShadowStackPush8B(tempSSP);
    FI;
    IF EndbranchEnabled(CPL)
        IF CPL = 3
            THEN
                IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
                IA32_U_CET.SUPPRESS = 0
            ELSE
                IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                IA32_S_CET.SUPPRESS = 0
        FI;
    FI;
END;

CALL-GATE:
```

```
    IF call gate (DPL < CPL) or (RPL > DPL) or (CR4.FRED = 1)
        THEN #GP(call-gate selector); FI;
    IF call gate not present
        THEN #NP(call-gate selector); FI;
    IF call-gate code-segment selector is NULL
        THEN #GP(0); FI;
    IF call-gate code-segment selector index is outside descriptor table limits
        THEN #GP(call-gate code-segment selector); FI;
    Read call-gate code-segment descriptor;
    IF call-gate code-segment descriptor does not indicate a code segment
    or call-gate code-segment descriptor DPL > CPL
        THEN #GP(call-gate code-segment selector); FI;
    IF IA32_EFER.LMA = 1 AND (call-gate code-segment descriptor is
    not a 64-bit code segment or call-gate code-segment descriptor has both L-bit and D-bit set)
        THEN #GP(call-gate code-segment selector); FI;
    IF call-gate code segment not present
        THEN #NP(call-gate code-segment selector); FI;
    IF call-gate code segment is non-conforming and DPL < CPL
        THEN go to MORE-PRIVILEGE;
        ELSE go to SAME-PRIVILEGE;
    FI;
END;

MORE-PRIVILEGE:
    IF current TSS is 32-bit
        THEN
            TSSstackAddress := (new code-segment DPL * 8) + 4;
            IF (TSSstackAddress + 5) > current TSS limit
                THEN #TS(current TSS selector); FI;
            NewSS := 2 bytes loaded from (TSS base + TSSstackAddress + 4);
            NewESP := 4 bytes loaded from (TSS base + TSSstackAddress);
        ELSE
            IF current TSS is 16-bit
                THEN
                    TSSstackAddress := (new code-segment DPL * 4) + 2
                    IF (TSSstackAddress + 3) > current TSS limit
                        THEN #TS(current TSS selector); FI;
                    NewSS := 2 bytes loaded from (TSS base + TSSstackAddress + 2);
                    NewESP := 2 bytes loaded from (TSS base + TSSstackAddress);
                ELSE (* current TSS is 64-bit *)
                    TSSstackAddress := (new code-segment DPL * 8) + 4;
                    IF (TSSstackAddress + 7) > current TSS limit
                        THEN #TS(current TSS selector); FI;
                    NewSS := new code-segment DPL; (* NULL selector with RPL = new CPL *)
                    NewRSP := 8 bytes loaded from (current TSS base + TSSstackAddress);
            FI;
    FI;
    IF IA32_EFER.LMA = 0 and NewSS is NULL
        THEN #TS(NewSS); FI;
    Read new stack-segment descriptor;
    IF IA32_EFER.LMA = 0 and (NewSS RPL ≠ new code-segment DPL
    or new stack-segment DPL ≠ new code-segment DPL or new stack segment is not a
    writable data segment)
        THEN #TS(NewSS); FI
```

```
IF IA32_EFER.LMA = 0 and new stack segment not present
    THEN #SS(NewSS); FI;
IF CallGateSize = 32
    THEN
        IF new stack does not have room for parameters plus 16 bytes
            THEN #SS(NewSS); FI;
        IF CallGate(InstructionPointer) not within new code-segment limit
            THEN #GP(0); FI;
        SS := newSS;  (* Segment descriptor information also loaded *)
        ESP := newESP;
        CS:EIP := CallGate(CS:InstructionPointer);
        (* Segment descriptor information also loaded *)
        Push(oldSS:oldESP); (* From calling procedure *)
        temp := parameter count from call gate, masked to 5 bits;
        Push(parameters from calling procedure's stack, temp)
        Push(oldCS:oldEIP); (* Return address to calling procedure *)
    ELSE
        IF CallGateSize = 16
            THEN
                IF new stack does not have room for parameters plus 8 bytes
                    THEN #SS(NewSS); FI;
                IF (CallGate(InstructionPointer) AND FFFFH) not in new code-segment limit
                    THEN #GP(0); FI;
                SS := newSS;  (* Segment descriptor information also loaded *)
                ESP := newESP;
                CS:IP := CallGate(CS:InstructionPointer);
                (* Segment descriptor information also loaded *)
                Push(oldSS:oldESP); (* From calling procedure *)
                temp := parameter count from call gate, masked to 5 bits;
                Push(parameters from calling procedure's stack, temp)
                Push(oldCS:oldEIP); (* Return address to calling procedure *)
            ELSE (* CallGateSize = 64 *)
                IF pushing 32 bytes on the stack would use a non-canonical address
                    THEN #SS(NewSS); FI;
                IF (CallGate(InstructionPointer) is non-canonical)
                    THEN #GP(0); FI;
                SS := NewSS; (* NewSS is NULL)
                RSP := NewESP;
                CS:IP := CallGate(CS:InstructionPointer);
                (* Segment descriptor information also loaded *)
                Push(oldSS:oldESP); (* From calling procedure *)
                Push(oldCS:oldEIP); (* Return address to calling procedure *)
        FI;
FI;
IF ShadowStackEnabled(CPL) AND CPL = 3
    THEN
        IF IA32_EFER.LMA = 0
            THEN IA32_PL3_SSP := SSP;
            ELSE (* adjust so bits 63:N get the value of bit N-1, where N is the CPU's maximum linear-address width *)
                IA32_PL3_SSP := LA_adjust(SSP);
        FI;
FI;
CPL := CodeSegment(DPL)
CS(RPL) := CPL
```

```
    IF ShadowStackEnabled(CPL)
        oldSSP := SSP
        SSP := IA32_PLi_SSP; (* where i is the CPL *)
        IF SSP & 0x07 != 0 (* if SSP not aligned to 8 bytes then #GP *)
            THEN #GP(0); FI;
        (* Token and CS:LIP:oldSSP pushed on shadow stack must be contained in a naturally aligned 32-byte region*)
        IF (SSP & ~0x1F) != ((SSP – 24) & ~0x1F)
            #GP(0); FI;
        IF ((IA32_EFER.LMA and CS.L) = 0 AND SSP[63:32] != 0)
            THEN #GP(0); FI;
        expected_token_value = SSP              (* busy bit - bit position 0 - must be clear *)
        new_token_value = SSP | BUSY_BIT        (* Set the busy bit *)
        IF shadow_stack_lock_cmpxchg8b(SSP, new_token_value, expected_token_value) != expected_token_value
            THEN #GP(0); FI;
        IF oldSS.DPL != 3
            ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits of 0 *)
            ShadowStackPush8B(oldCSBASE+oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
            ShadowStackPush8B(oldSSP);
        FI;
    FI;
    IF EndbranchEnabled (CPL)
        IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
        IA32_S_CET.SUPPRESS = 0
    FI;
END;

SAME-PRIVILEGE:
    IF CallGateSize = 32
        THEN
            IF stack does not have room for 8 bytes
                THEN #SS(0); FI;
            IF CallGate(InstructionPointer) not within code segment limit
                THEN #GP(0); FI;
            CS:EIP := CallGate(CS:EIP) (* Segment descriptor information also loaded *)
            Push(oldCS:oldEIP); (* Return address to calling procedure *)
        ELSE
            If CallGateSize = 16
                THEN
                    IF stack does not have room for 4 bytes
                        THEN #SS(0); FI;
                    IF CallGate(InstructionPointer) not within code segment limit
                        THEN #GP(0); FI;
                    CS:IP := CallGate(CS:instruction pointer);
                    (* Segment descriptor information also loaded *)
                    Push(oldCS:oldIP); (* Return address to calling procedure *)
                ELSE (* CallGateSize = 64)
                    IF pushing 16 bytes on the stack touches non-canonical addresses
                        THEN #SS(0); FI;
                    IF RIP non-canonical
                        THEN #GP(0); FI;
                    CS:IP := CallGate(CS:instruction pointer);
                    (* Segment descriptor information also loaded *)
                    Push(oldCS:oldIP); (* Return address to calling procedure *)
            FI;
```

```
    FI;
    CS(RPL) := CPL
    IF ShadowStackEnabled(CPL)
        (* Align to next 8 byte boundary *)
        tempSSP = SSP;
        Shadow_stack_store 4 bytes of 0 to (SSP – 4)
        SSP = SSP & 0xFFFFFFFFFFFFFFF8H;
        (* push cs:lip:ssp on shadow stack *)
        ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits of 0 *)
        ShadowStackPush8B(oldCSBASE + oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
        ShadowStackPush8B(tempSSP);
    FI;
    IF EndbranchEnabled (CPL)
        IF CPL = 3
            THEN
                IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH;
                IA32_U_CET.SUPPRESS = 0
            ELSE
                IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH;
                IA32_S_CET.SUPPRESS = 0
        FI;
    FI;
END;

TASK-GATE:
    IF task gate DPL < CPL or RPL
        THEN #GP(task gate selector); FI;
    IF task gate not present
        THEN #NP(task gate selector); FI;
    Read the TSS segment selector in the task-gate descriptor;
    IF TSS segment selector local/global bit is set to local
    or index not within GDT limits
        THEN #GP(TSS selector); FI;
    Access TSS descriptor in GDT;
    IF descriptor is not a TSS segment
        THEN #GP(TSS selector); FI;
    IF TSS descriptor specifies that the TSS is busy
        THEN #GP(TSS selector); FI;
    IF TSS not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS (with nesting) to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;

TASK-STATE-SEGMENT:
    IF TSS DPL < CPL or RPL
    or TSS descriptor indicates TSS not available
        THEN #GP(TSS selector); FI;
    IF TSS is not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS (with nesting) to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
```

```
END;
```

Flags Affected

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

Protected Mode Exceptions

#GP(0)	If the target offset in destination operand is beyond the new code segment limit.
	If the segment selector in the destination operand is NULL.
	If the code segment selector in the gate is NULL.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If target mode is compatibility mode and SSP is not in low 4GB.
	If SSP in IA32_PLi_SSP (where i is the new CPL) is not 8 byte aligned.
	If the token and the stack frame to be pushed on shadow stack are not contained in a naturally aligned 32-byte region of the shadow stack.
	If "supervisor Shadow Stack" token on new shadow stack is marked busy.
	If destination mode is 32-bit or compatibility mode, but SSP address in "supervisor shadow stack" token is beyond 4GB.
	If SSP address in "supervisor shadow stack" token does not match SSP address in IA32_PLi_SSP (where i is the new CPL).
#GP(selector)	If a code segment or gate or TSS selector index is outside descriptor table limits.
	If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment.
	If the DPL for a nonconforming-code segment is not equal to the CPL or the RPL for the segment's segment selector is greater than the CPL.
	If the DPL for a conforming-code segment is greater than the CPL.
	If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS's segment selector.
	If the segment descriptor for a segment selector from a call gate does not indicate it is a code segment.
	If the segment selector from a call gate is beyond the descriptor table limits.
	If the DPL for a code-segment obtained from a call gate is greater than the CPL.
	If the segment selector for a TSS has its local/global bit set for local.
	If a TSS segment descriptor specifies that the TSS is busy or not available.
#SS(0)	If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when no stack switch occurs.
	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when a stack switch occurs.
	If the SS register is being loaded as part of a stack switch and the segment pointed to is marked not present.
	If stack segment does not have room for the return address, parameters, or stack segment pointer, when stack switch occurs.
#NP(selector)	If a code segment, data segment, call gate, task gate, or TSS is not present.

#TS(selector)	If the new stack segment selector and ESP are beyond the end of the TSS.
	If the new stack segment selector is NULL.
	If the RPL of the new stack segment selector in the TSS is not equal to the DPL of the code segment being accessed.
	If DPL of the stack segment descriptor for the new stack segment is not equal to the DPL of the code segment descriptor.
	If the new stack segment is not a writable data segment.
	If segment-selector index for stack segment is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

#GP(selector)	If a memory address accessed by the selector is in non-canonical space.
	If FRED transitions are enabled and the selector references a call gate.
#GP(0)	If the target offset in the destination operand is non-canonical.

64-Bit Mode Exceptions

#GP(0)	If a memory address is non-canonical.
	If target offset in destination operand is non-canonical.
	If the segment selector in the destination operand is NULL.
	If the code segment selector in the 64-bit gate is NULL.
	If target mode is compatibility mode and SSP is not in low 4GB.
	If SSP in IA32_PLi_SSP (where i is the new CPL) is not 8 byte aligned.
	If the token and the stack frame to be pushed on shadow stack are not contained in a naturally aligned 32-byte region of the shadow stack.
	If “supervisor Shadow Stack” token on new shadow stack is marked busy.
	If destination mode is 32-bit mode or compatibility mode, but SSP address in “super-visor shadow” stack token is beyond 4GB.
	If SSP address in “supervisor shadow stack” token does not match SSP address in IA32_PLi_SSP (where i is the new CPL).

#GP(selector)	If code segment or 64-bit call gate is outside descriptor table limits.
	If code segment or 64-bit call gate overlaps non-canonical space.
	If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, or 64-bit call gate.
	If the segment descriptor pointed to by the segment selector in the destination operand is a code segment and has both the D-bit and the L- bit set.
	If the DPL for a nonconforming-code segment is not equal to the CPL, or the RPL for the segment’s segment selector is greater than the CPL.
	If the DPL for a conforming-code segment is greater than the CPL.
	If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate.
	If the upper type field of a 64-bit call gate is not 0x0.
	If the segment selector from a 64-bit call gate is beyond the descriptor table limits.
	If the DPL for a code-segment obtained from a 64-bit call gate is greater than the CPL.
	If the code segment descriptor pointed to by the selector in the 64-bit gate doesn't have the L-bit set and the D-bit clear.
	If the segment descriptor for a segment selector from the 64-bit call gate does not indicate it is a code segment.
	If FRED transitions are enabled and the selector references a call gate.
	If FRED transitions are enabled, CPL is 0, and the instruction would enter compatibility mode.
#SS(0)	If pushing the return offset or CS selector onto the stack exceeds the bounds of the stack segment when no stack switch occurs.
	If a memory operand effective address is outside the SS segment limit.
	If the stack address is in a non-canonical form.
#SS(selector)	If pushing the old values of SS selector, stack pointer, EFLAGS, CS selector, offset, or error code onto the stack violates the canonical boundary when a stack switch occurs.
#NP(selector)	If a code segment or 64-bit call gate is not present.
#TS(selector)	If the load of the new RSP exceeds the limit of the TSS.
#UD	(64-bit mode only) If a far call is direct to an absolute address in memory.
	If the LOCK prefix is used.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

CBW/CWDE/CDQE—Convert Byte to Word/Convert Word to Doubleword/Convert Doubleword to Quadword

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
98	CBW	ZO	Valid	Valid	AX := sign-extend of AL.
98	CWDE	ZO	Valid	Valid	EAX := sign-extend of AX.
REX.W + 98	CDQE	ZO	Valid	N.E.	RAX := sign-extend of EAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Double the size of the source operand by means of sign extension. The CBW (convert byte to word) instruction copies the sign (bit 7) in the source operand into every bit in the AH register. The CWDE (convert word to double-word) instruction copies the sign (bit 15) of the word in the AX register into the high 16 bits of the EAX register.

CBW and CWDE reference the same opcode. The CBW instruction is intended for use when the operand-size attribute is 16; CWDE is intended for use when the operand-size attribute is 32. Some assemblers may force the operand size. Others may treat these two mnemonics as synonyms (CBW/CWDE) and use the setting of the operand-size attribute to determine the size of values to be converted.

In 64-bit mode, the default operation size is the size of the destination register. Use of the REX.W prefix promotes this instruction (CDQE when promoted) to operate on 64-bit operands. In which case, CDQE copies the sign (bit 31) of the doubleword in the EAX register into the high 32 bits of RAX.

Operation

```
IF OperandSize = 16 (* Instruction = CBW *)
    THEN
        AX := SignExtend(AL);
    ELSE IF (OperandSize = 32, Instruction = CWDE)
        EAX := SignExtend(AX); FI;
    ELSE (* 64-Bit Mode, OperandSize = 64, Instruction = CDQE*)
        RAX := SignExtend(EAX);
FI;
```

Flags Affected

None.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

CLAC—Clear AC Flag in EFLAGS Register

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 CA CLAC	ZO	V/V	SMAP	Clear the AC flag in the EFLAGS register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Clears the AC flag bit in EFLAGS register. This disables any alignment checking of user-mode data accesses. If the SMAP bit is set in the CR4 register, this disallows explicit supervisor-mode data accesses to user-mode pages.

This instruction's operation is the same in non-64-bit modes and 64-bit mode. Attempts to execute CLAC when CPL > 0 cause #UD.

Operation

```
EFLAGS.AC := 0;
```

Flags Affected

AC cleared. Other flags are unaffected.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If the CPL > 0.
	If CPUID.07H.00H:EBX.SMAP[20] = 0.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.07H.00H:EBX.SMAP[20] = 0.

Virtual-8086 Mode Exceptions

#UD	The CLAC instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.
	If the CPL > 0.
	If CPUID.07H.00H:EBX.SMAP[20] = 0.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If the CPL > 0.
	If CPUID.07H.00H:EBX.SMAP[20] = 0.

CLC—Clear Carry Flag

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
F8	CLC	ZO	Valid	Valid	Clear CF flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Clears the CF flag in the EFLAGS register. Operation is the same in all modes.

Operation

```
CF := 0;
```

Flags Affected

The CF flag is set to 0. The OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

CLD—Clear Direction Flag

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
FC	CLD	ZO	Valid	Valid	Clear DF flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Clears the DF flag in the EFLAGS register. When the DF flag is set to 0, string operations increment the index registers (ESI and/or EDI). Operation is the same in all modes.

Operation

```
DF := 0;
```

Flags Affected

The DF flag is set to 0. The CF, OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

CLDEMOTE—Cache Line Demote

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 1C /0 CLDEMOTE m8	A	V/V	CLDEMOTE	Hint to hardware to move the cache line containing m8 to a more distant level of the cache without writing back to memory.

Instruction Operand Encoding[1]

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

Description

Hints to hardware that the cache line that contains the linear address specified with the memory operand should be moved ("demoted") from the cache(s) closest to the processor core to a level more distant from the processor core. This may accelerate subsequent accesses to the line by other cores in the same coherence domain, especially if the line was written by the core that demotes the line. Moving the line in such a manner is a performance optimization, i.e., it is a hint which does not modify architectural state. Hardware may choose which level in the cache hierarchy to retain the line (e.g., L3 in typical server designs). The source operand is a byte memory location.

The availability of the CLDEMOTE instruction is indicated by the presence of the CPUID feature flag CLDEMOTE (bit 25 of the ECX register in sub-leaf 07H, see "CPUID—CPU Identification"). On processors which do not support the CLDEMOTE instruction (including legacy hardware) the instruction will be treated as a NOP.

A CLDEMOTE instruction is ordered with respect to stores to the same cache line, but unordered with respect to other instructions including memory fences, CLDEMOTE, CLWB or CLFLUSHOPT instructions to a different cache line. Since CLDEMOTE will retire in order with respect to stores to the same cache line, software should ensure that after issuing CLDEMOTE the line is not accessed again immediately by the same core to avoid cache data movement penalties.

The effective memory type of the page containing the affected line determines the effect; cacheable types are likely to generate a data movement operation, while uncacheable types may cause the instruction to be ignored.

Speculative fetching can occur at any time and is not tied to instruction execution. The CLDEMOTE instruction is not ordered with respect to PREFETCHh instructions or any of the speculative fetching mechanisms. That is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLDEMOTE instruction that references the cache line.

Unlike CLFLUSH, CLFLUSHOPT, and CLWB instructions, CLDEMOTE is not guaranteed to write back modified data to memory.

The CLDEMOTE instruction may be ignored by hardware in certain cases and is not a guarantee.

The CLDEMOTE instruction can be used at all privilege levels. In certain processor implementations the CLDEMOTE instruction may set the A bit but not the D bit in the page tables.

If the line is not found in the cache, the instruction will be treated as a NOP.

In some implementations, the CLDEMOTE instruction may always cause a transactional abort with Transactional Synchronization Extensions (TSX). However, programmers must not rely on CLDEMOTE instruction to force a transactional abort.

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

Operation

```
Cache_Line_Demote(m8);
```

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

CLDEMOTE void _cldemote(const 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

Same exceptions as in real address mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.

CLFLUSH—Flush Cache Line

Opcode / Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
NP 0F AE /7 CLFLUSH *m8*	M	Valid	Valid	Flushes cache line containing *m8*.

Instruction Operand Encoding

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

Description

Invalidates from every level of the cache hierarchy in the cache coherence domain the cache line that contains the linear address specified with the memory operand. If that cache line contains modified data at any level of the cache hierarchy, that data is written back to memory. The source operand is a byte memory location.

The availability of CLFLUSH is indicated by the presence of the CPUID feature flag CLFLUSH (CPUID.01H:EDX[19]). The aligned cache line size affected is also indicated by the value in CPUID.01H:EBX[15:8].

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions assigned a memory-type allowing for speculative reads (such as, the WB, WC, and WT memory types). PREFETCH*h* instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLFLUSH instruction is not ordered with respect to PREFETCH*h* instructions or any of the speculative fetching mechanisms (that is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLFLUSH instruction that references the cache line).

Executions of the CLFLUSH instruction are ordered with respect to each other and with respect to writes, locked read-modify-write instructions, and fence instructions.[1] They are not ordered with respect to executions of CLFLUSHOPT and CLWB. Software can use the SFENCE instruction to order an execution of CLFLUSH relative to one of those operations.

The CLFLUSH instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load (and in addition, a CLFLUSH instruction is allowed to flush a linear address in an execute-only segment). Like a load, the CLFLUSH instruction sets the A bit but not the D bit in the page tables.

In some implementations, the CLFLUSH instruction may always cause transactional abort with Transactional Synchronization Extensions (TSX). The CLFLUSH instruction is not expected to be commonly used inside typical transactional regions. However, programmers must not rely on CLFLUSH instruction to force a transactional abort, since whether they cause transactional abort is implementation dependent.

The CLFLUSH instruction was introduced with the SSE2 extensions; however, because it has its own CPUID feature flag, it can be implemented in IA-32 processors that do not include the SSE2 extensions. Also, detecting the presence of the SSE2 extensions with the CPUID instruction does not guarantee that the CLFLUSH instruction is implemented in the processor.

CLFLUSH operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
Flush_Cache_Line(SRC);
```

Intel C/C++ Compiler Intrinsic Equivalents

CLFLUSH void _mm_clflush(void const *p)

1. Earlier versions of this manual specified that executions of the CLFLUSH instruction were ordered only by the MFENCE instruction. All processors implementing the CLFLUSH instruction also order it relative to the other operations enumerated above.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:EDX.CLFLUSH[19] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD	If CPUID.01H:EDX.CLFLUSH[19] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:EDX.CLFLUSH[19] = 0.
	If the LOCK prefix is used.

CLFLUSHOPT—Flush Cache Line Optimized

Opcode / Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
NFx 66 0F AE /7 CLFLUSHOPT m8	M	Valid	Valid	Flushes cache line containing m8.

Instruction Operand Encoding

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

Description

Invalidates from every level of the cache hierarchy in the cache coherence domain the cache line that contains the linear address specified with the memory operand. If that cache line contains modified data at any level of the cache hierarchy, that data is written back to memory. The source operand is a byte memory location.

The availability of CLFLUSHOPT is indicated by the presence of the CPUID feature flag CLFLUSHOPT (CPUID.07H.00H:EBX[23]). The aligned cache line size affected is also indicated by the value in CPUID.01H:EBX[15:8].

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions assigned a memory-type allowing for speculative reads (such as, the WB, WC, and WT memory types). PREFETCH*h* instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLFLUSH instruction is not ordered with respect to PREFETCH*h* instructions or any of the speculative fetching mechanisms (that is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLFLUSH instruction that references the cache line).

Executions of the CLFLUSHOPT instruction are ordered with respect to fence instructions and to locked read-modify-write instructions; they are also ordered with respect to older writes to the cache line being invalidated. They are not ordered with respect to other executions of CLFLUSHOPT, to executions of CLFLUSH and CLWB, or to younger writes to the cache line being invalidated. Software can use the SFENCE instruction to order an execution of CLFLUSHOPT relative to one of those operations.

The CLFLUSHOPT instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load (and in addition, a CLFLUSHOPT instruction is allowed to flush a linear address in an execute-only segment). Like a load, the CLFLUSHOPT instruction sets the A bit but not the D bit in the page tables.

In some implementations, the CLFLUSHOPT instruction may always cause transactional abort with Transactional Synchronization Extensions (TSX). The CLFLUSHOPT instruction is not expected to be commonly used inside typical transactional regions. However, programmers must not rely on CLFLUSHOPT instruction to force a transactional abort, since whether they cause transactional abort is implementation dependent.

CLFLUSHOPT operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
Flush_Cache_Line_Optimized(SRC);
```

Intel C/C++ Compiler Intrinsic Equivalents

CLFLUSHOPT void _mm_clflushopt(void const *p)

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#UD	If CPUID.07H.00H:EBX.CLFLUSHOPT[23] = 0.
	If the LOCK prefix is used.
	If an instruction prefix F2H or F3H is used.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD	If CPUID.07H.00H:EBX.CLFLUSHOPT[23] = 0.
	If the LOCK prefix is used.
	If an instruction prefix F2H or F3H is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#UD	If CPUID.07H.00H:EBX.CLFLUSHOPT[23] = 0.
	If the LOCK prefix is used.
	If an instruction prefix F2H or F3H is used.

CLI—Clear Interrupt Flag

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
FA	CLI	ZO	Valid	Valid	Clear interrupt flag; interrupts disabled when interrupt flag cleared.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

In most cases, CLI clears the IF flag in the EFLAGS register and no other flags are affected. Clearing the IF flag causes the processor to ignore maskable external interrupts. The IF flag and the CLI and STI instruction have no effect on the generation of exceptions and NMI interrupts.

Operation is different in two modes defined as follows:

- **PVI mode** (protected-mode virtual interrupts): CR0.PE = 1, EFLAGS.VM = 0, CPL = 3, and CR4.PVI = 1;
- **VME mode** (virtual-8086 mode extensions): CR0.PE = 1, EFLAGS.VM = 1, and CR4.VME = 1.

If IOPL < 3 and either VME mode or PVI mode is active, CLI clears the VIF flag in the EFLAGS register, leaving IF unaffected.

Table 3-7 indicates the action of the CLI instruction depending on the processor operating mode, IOPL, and CPL.

Table 3-7. Decision Table for CLI Results

Mode	IOPL	CLI Result
Real-address	X[1]	IF = 0
Protected, not PVI[2]	≥ CPL	IF = 0
	< CPL	#GP fault
Protected, PVI[3]	3	IF = 0
	0–2	VIF = 0
Virtual-8086, not VME[3]	3	IF = 0
	0–2	#GP fault
Virtual-8086, VME[3]	3	IF = 0
	0–2	VIF = 0

NOTES:

1. X = This setting has no effect on instruction operation.
2. For this table, "protected mode" applies whenever CR0.PE = 1 and EFLAGS.VM = 0; it includes compatibility mode and 64-bit mode.
3. PVI mode and virtual-8086 mode each imply CPL = 3.

Operation

```
IF CR0.PE = 0
    THEN IF := 0; (* Reset Interrupt Flag *)
    ELSE
        IF IOPL ≥ CPL  (* CPL = 3 if EFLAGS.VM = 1 *)
            THEN IF := 0; (* Reset Interrupt Flag *)
            ELSE
                IF VME mode OR PVI mode
                    THEN VIF := 0; (* Reset Virtual Interrupt Flag *)
                    ELSE #GP(0);
                FI;
        FI;
FI;
```

Flags Affected

Either the IF flag or the VIF flag is cleared to 0. Other flags are unaffected.

Protected Mode Exceptions

#GP(0)	If CPL is greater than IOPL and PVI mode is not active.
	If CPL is greater than IOPL and less than 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If IOPL is less than 3 and VME mode is not active.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

CLRSSBSY—Clear Busy Flag in a Supervisor Shadow Stack Token

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F AE /6 CLRSSBSY m64	M	V/V	CET_SS	Clear busy flag in supervisor shadow stack token reference by m64.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
M	N/A	ModRM:r/m (r, w)	N/A	N/A	N/A

Description

Clear busy flag in supervisor shadow stack token reference by m64. Subsequent to marking the shadow stack as not busy the SSP is loaded with value 0.

This instruction cannot be executed when FRED transitions are enabled. FRED transitions do not use supervisor shadow stack tokens.

Operation

```
IF CR4.CET = 0 OR CR4.FRED = 1
    THEN #UD; FI;

IF IA32_S_CET.SH_STK_EN = 0
    THEN #UD; FI;

IF CPL > 0
    THEN GP(0); FI;

SSP_LA = Linear_Address(mem operand)
IF SSP_LA not aligned to 8 bytes
    THEN #GP(0); FI;
expected_token_value = SSP_LA | BUSY_BIT  (* busy bit - bit position 0 - must be set *)
new_token_value = SSP_LA                  (* Clear the busy bit *)
IF shadow_stack_lock_cmpxchg8b(SSP_LA, new_token_value, expected_token_value) != expected_token_value
    invalid_token := 1; FI

(* Set the CF if invalid token was detected *)
RFLAGS.CF = (invalid_token == 1) ? 1 : 0;
RFLAGS.ZF,PF,AF,OF,SF := 0;
SSP := 0
```

Flags Affected

CF is set if an invalid token was detected, else it is cleared. ZF, PF, AF, OF, and SF are cleared.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	IF IA32_S_CET.SH_STK_EN = 0.
#GP(0)	If memory operand linear address not aligned to 8 bytes.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If destination is located in a non-writable segment.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If CPL is not 0.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	The CLRSSBSY instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The CLRSSBSY instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	IF IA32_S_CET.SH_STK_EN = 0.
	If CR4.FRED = 1.
#GP(0)	If memory operand linear address not aligned to 8 bytes.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If destination is located in a non-writable segment.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If CPL is not 0.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	If CR4.FRED = 1.
	IF IA32_S_CET.SH_STK_EN = 0.
#GP(0)	If memory operand linear address not aligned to 8 bytes.
	If CPL is not 0.
	If the memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.

CLTS—Clear Task-Switched Flag in CR0

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
0F 06	CLTS	ZO	Valid	Valid	Clears TS flag in CR0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Clears the task-switched (TS) flag in the CR0 register. This instruction is intended for use in operating-system procedures. It is a privileged instruction that can only be executed at a CPL of 0. It is allowed to be executed in real-address mode to allow initialization for protected mode.

The processor sets the TS flag every time a task switch occurs. The flag is used to synchronize the saving of FPU context in multitasking applications. See the description of the TS flag in the section titled "Control Registers" in Chapter 2 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, for more information about this flag.

CLTS operation is the same in non-64-bit modes and 64-bit mode.

See Chapter 27, "Virtual Machine Control Structures," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C, for more information about the behavior of this instruction in VMX non-root operation.

Operation

```
CR0.TS[bit 3] := 0;
```

Flags Affected

The TS flag in CR0 register is cleared.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	CLTS is not recognized in virtual-8086 mode.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the CPL is greater than 0.
#UD	If the LOCK prefix is used.

CLUI—Clear User Interrupt Flag

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 01 EE CLUI	ZO	V/I	UINTR	Clear user interrupt flag; user interrupts blocked when user interrupt flag cleared.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A	N/A

Description

CLUI clears the user interrupt flag (UIF). Its effect takes place immediately: a user interrupt cannot be delivered on the instruction boundary following CLUI.

An execution of CLUI inside a transactional region causes a transactional abort; the abort loads EAX as it would have had it been caused due to an execution of CLI.

Operation

```
UIF := 0;
```

Flags Affected

None.

Protected Mode Exceptions

#UD	The CLUI instruction is not recognized in protected mode.

Real-Address Mode Exceptions

#UD	The CLUI instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The CLUI instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	The CLUI instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used. If executed inside an enclave. If CR4.UINTR = 0. If CPUID.07H.00H:EDX.UINTR[5] = 0.

CLWB—Cache Line Write Back

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F AE /6 CLWB m8	M	V/V	CLWB	Writes back modified cache line containing m8, and may retain the line in cache hierarchy in non-modified state.

Instruction Operand Encoding[1]

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

Description

Writes back to memory the cache line (if modified) that contains the linear address specified with the memory operand from any level of the cache hierarchy in the cache coherence domain. The line may be retained in the cache hierarchy in non-modified state. Retaining the line in the cache hierarchy is a performance optimization (treated as a hint by hardware) to reduce the possibility of cache miss on a subsequent access. Hardware may choose to retain the line at any of the levels in the cache hierarchy, and in some cases, may invalidate the line from the cache hierarchy. The source operand is a byte memory location.

The availability of CLWB instruction is indicated by CPUID.07H.00H:EBX.CLWB[24]. The aligned cache line size affected is also indicated by the value in CPUID.01H:EBX[15:8].

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type allowing for speculative reads (such as, the WB, WC, and WT memory types). PREFETCHh instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLWB instruction is not ordered with respect to PREFETCHh instructions or any of the speculative fetching mechanisms (that is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLWB instruction that references the cache line).

Executions of the CLWB instruction are ordered with respect to fence instructions and to locked read-modify-write instructions; they are also ordered with respect to older writes to the cache line being written back. They are not ordered with respect to other executions of CLWB, to executions of CLFLUSH and CLFLUSHOPT, or to younger writes to the cache line being written back. Software can use the SFENCE instruction to order an execution of CLWB relative to one of those operations.

For usages that require only writing back modified data from cache lines to memory (do not require the line to be invalidated), and expect to subsequently access the data, software is recommended to use CLWB (with appropriate fencing) instead of CLFLUSH or CLFLUSHOPT for improved performance.

The CLWB instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load. Like a load, the CLWB instruction sets the accessed flag but not the dirty flag in the page tables.

In some implementations, the CLWB instruction may always cause transactional abort with Transactional Synchronization Extensions (TSX). CLWB instruction is not expected to be commonly used inside typical transactional regions. However, programmers must not rely on CLWB instruction to force a transactional abort, since whether they cause transactional abort is implementation dependent.

Operation

```
Cache_Line_Write_Back(m8);
```

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

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

CLWB void _mm_clwb(void const *p);

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.07H.00H:EBX.CLWB[24] = 0.
#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.07H.00H:EBX.CLWB[24] = 0.
#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.07H.00H:EBX.CLWB[24] = 0.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.

CMC—Complement Carry Flag

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
F5	CMC	ZO	Valid	Valid	Complement CF flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Complements the CF flag in the EFLAGS register. CMC operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
EFLAGS.CF[bit 0] := NOT EFLAGS.CF[bit 0];
```

Flags Affected

The CF flag contains the complement of its original value. The OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

CMOV*cc*—Conditional Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 47 /r	CMOVA r16, r/m16	RM	Valid	Valid	Move if above (CF=0 and ZF=0).
0F 47 /r	CMOVA r32, r/m32	RM	Valid	Valid	Move if above (CF=0 and ZF=0).
REX.W + 0F 47 /r	CMOVA r64, r/m64	RM	Valid	N.E.	Move if above (CF=0 and ZF=0).
0F 43 /r	CMOVAE r16, r/m16	RM	Valid	Valid	Move if above or equal (CF=0).
0F 43 /r	CMOVAE r32, r/m32	RM	Valid	Valid	Move if above or equal (CF=0).
REX.W + 0F 43 /r	CMOVAE r64, r/m64	RM	Valid	N.E.	Move if above or equal (CF=0).
0F 42 /r	CMOVB r16, r/m16	RM	Valid	Valid	Move if below (CF=1).
0F 42 /r	CMOVB r32, r/m32	RM	Valid	Valid	Move if below (CF=1).
REX.W + 0F 42 /r	CMOVB r64, r/m64	RM	Valid	N.E.	Move if below (CF=1).
0F 46 /r	CMOVBE r16, r/m16	RM	Valid	Valid	Move if below or equal (CF=1 or ZF=1).
0F 46 /r	CMOVBE r32, r/m32	RM	Valid	Valid	Move if below or equal (CF=1 or ZF=1).
REX.W + 0F 46 /r	CMOVBE r64, r/m64	RM	Valid	N.E.	Move if below or equal (CF=1 or ZF=1).
0F 42 /r	CMOVC r16, r/m16	RM	Valid	Valid	Move if carry (CF=1).
0F 42 /r	CMOVC r32, r/m32	RM	Valid	Valid	Move if carry (CF=1).
REX.W + 0F 42 /r	CMOVC r64, r/m64	RM	Valid	N.E.	Move if carry (CF=1).
0F 44 /r	CMOVE r16, r/m16	RM	Valid	Valid	Move if equal (ZF=1).
0F 44 /r	CMOVE r32, r/m32	RM	Valid	Valid	Move if equal (ZF=1).
REX.W + 0F 44 /r	CMOVE r64, r/m64	RM	Valid	N.E.	Move if equal (ZF=1).
0F 4F /r	CMOVG r16, r/m16	RM	Valid	Valid	Move if greater (ZF=0 and SF=OF).
0F 4F /r	CMOVG r32, r/m32	RM	Valid	Valid	Move if greater (ZF=0 and SF=OF).
REX.W + 0F 4F /r	CMOVG r64, r/m64	RM	V/N.E.	N/A	Move if greater (ZF=0 and SF=OF).
0F 4D /r	CMOVGE r16, r/m16	RM	Valid	Valid	Move if greater or equal (SF=OF).
0F 4D /r	CMOVGE r32, r/m32	RM	Valid	Valid	Move if greater or equal (SF=OF).
REX.W + 0F 4D /r	CMOVGE r64, r/m64	RM	Valid	N.E.	Move if greater or equal (SF=OF).
0F 4C /r	CMOVL r16, r/m16	RM	Valid	Valid	Move if less (SF≠ OF).
0F 4C /r	CMOVL r32, r/m32	RM	Valid	Valid	Move if less (SF≠ OF).
REX.W + 0F 4C /r	CMOVL r64, r/m64	RM	Valid	N.E.	Move if less (SF≠ OF).
0F 4E /r	CMOVLE r16, r/m16	RM	Valid	Valid	Move if less or equal (ZF=1 or SF≠ OF).
0F 4E /r	CMOVLE r32, r/m32	RM	Valid	Valid	Move if less or equal (ZF=1 or SF≠ OF).
REX.W + 0F 4E /r	CMOVLE r64, r/m64	RM	Valid	N.E.	Move if less or equal (ZF=1 or SF≠ OF).
0F 46 /r	CMOVNA r16, r/m16	RM	Valid	Valid	Move if not above (CF=1 or ZF=1).
0F 46 /r	CMOVNA r32, r/m32	RM	Valid	Valid	Move if not above (CF=1 or ZF=1).
REX.W + 0F 46 /r	CMOVNA r64, r/m64	RM	Valid	N.E.	Move if not above (CF=1 or ZF=1).
0F 42 /r	CMOVNAE r16, r/m16	RM	Valid	Valid	Move if not above or equal (CF=1).
0F 42 /r	CMOVNAE r32, r/m32	RM	Valid	Valid	Move if not above or equal (CF=1).
REX.W + 0F 42 /r	CMOVNAE r64, r/m64	RM	Valid	N.E.	Move if not above or equal (CF=1).
0F 43 /r	CMOVNB r16, r/m16	RM	Valid	Valid	Move if not below (CF=0).
0F 43 /r	CMOVNB r32, r/m32	RM	Valid	Valid	Move if not below (CF=0).
REX.W + 0F 43 /r	CMOVNB r64, r/m64	RM	Valid	N.E.	Move if not below (CF=0).
0F 47 /r	CMOVNBE r16, r/m16	RM	Valid	Valid	Move if not below or equal (CF=0 and ZF=0).

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 47 /r	CMOVNBE r32, r/m32	RM	Valid	Valid	Move if not below or equal (CF=0 and ZF=0).
REX.W + 0F 47 /r	CMOVNBE r64, r/m64	RM	Valid	N.E.	Move if not below or equal (CF=0 and ZF=0).
0F 43 /r	CMOVNC r16, r/m16	RM	Valid	Valid	Move if not carry (CF=0).
0F 43 /r	CMOVNC r32, r/m32	RM	Valid	Valid	Move if not carry (CF=0).
REX.W + 0F 43 /r	CMOVNC r64, r/m64	RM	Valid	N.E.	Move if not carry (CF=0).
0F 45 /r	CMOVNE r16, r/m16	RM	Valid	Valid	Move if not equal (ZF=0).
0F 45 /r	CMOVNE r32, r/m32	RM	Valid	Valid	Move if not equal (ZF=0).
REX.W + 0F 45 /r	CMOVNE r64, r/m64	RM	Valid	N.E.	Move if not equal (ZF=0).
0F 4E /r	CMOVNG r16, r/m16	RM	Valid	Valid	Move if not greater (ZF=1 or SF≠ OF).
0F 4E /r	CMOVNG r32, r/m32	RM	Valid	Valid	Move if not greater (ZF=1 or SF≠ OF).
REX.W + 0F 4E /r	CMOVNG r64, r/m64	RM	Valid	N.E.	Move if not greater (ZF=1 or SF≠ OF).
0F 4C /r	CMOVNGE r16, r/m16	RM	Valid	Valid	Move if not greater or equal (SF≠ OF).
0F 4C /r	CMOVNGE r32, r/m32	RM	Valid	Valid	Move if not greater or equal (SF≠ OF).
REX.W + 0F 4C /r	CMOVNGE r64, r/m64	RM	Valid	N.E.	Move if not greater or equal (SF≠ OF).
0F 4D /r	CMOVNL r16, r/m16	RM	Valid	Valid	Move if not less (SF=OF).
0F 4D /r	CMOVNL r32, r/m32	RM	Valid	Valid	Move if not less (SF=OF).
REX.W + 0F 4D /r	CMOVNL r64, r/m64	RM	Valid	N.E.	Move if not less (SF=OF).
0F 4F /r	CMOVNLE r16, r/m16	RM	Valid	Valid	Move if not less or equal (ZF=0 and SF=OF).
0F 4F /r	CMOVNLE r32, r/m32	RM	Valid	Valid	Move if not less or equal (ZF=0 and SF=OF).
REX.W + 0F 4F /r	CMOVNLE r64, r/m64	RM	Valid	N.E.	Move if not less or equal (ZF=0 and SF=OF).
0F 41 /r	CMOVNO r16, r/m16	RM	Valid	Valid	Move if not overflow (OF=0).
0F 41 /r	CMOVNO r32, r/m32	RM	Valid	Valid	Move if not overflow (OF=0).
REX.W + 0F 41 /r	CMOVNO r64, r/m64	RM	Valid	N.E.	Move if not overflow (OF=0).
0F 4B /r	CMOVNP r16, r/m16	RM	Valid	Valid	Move if not parity (PF=0).
0F 4B /r	CMOVNP r32, r/m32	RM	Valid	Valid	Move if not parity (PF=0).
REX.W + 0F 4B /r	CMOVNP r64, r/m64	RM	Valid	N.E.	Move if not parity (PF=0).
0F 49 /r	CMOVNS r16, r/m16	RM	Valid	Valid	Move if not sign (SF=0).
0F 49 /r	CMOVNS r32, r/m32	RM	Valid	Valid	Move if not sign (SF=0).
REX.W + 0F 49 /r	CMOVNS r64, r/m64	RM	Valid	N.E.	Move if not sign (SF=0).
0F 45 /r	CMOVNZ r16, r/m16	RM	Valid	Valid	Move if not zero (ZF=0).
0F 45 /r	CMOVNZ r32, r/m32	RM	Valid	Valid	Move if not zero (ZF=0).
REX.W + 0F 45 /r	CMOVNZ r64, r/m64	RM	Valid	N.E.	Move if not zero (ZF=0).
0F 40 /r	CMOVO r16, r/m16	RM	Valid	Valid	Move if overflow (OF=1).
0F 40 /r	CMOVO r32, r/m32	RM	Valid	Valid	Move if overflow (OF=1).
REX.W + 0F 40 /r	CMOVO r64, r/m64	RM	Valid	N.E.	Move if overflow (OF=1).
0F 4A /r	CMOVP r16, r/m16	RM	Valid	Valid	Move if parity (PF=1).
0F 4A /r	CMOVP r32, r/m32	RM	Valid	Valid	Move if parity (PF=1).
REX.W + 0F 4A /r	CMOVP r64, r/m64	RM	Valid	N.E.	Move if parity (PF=1).
0F 4A /r	CMOVPE r16, r/m16	RM	Valid	Valid	Move if parity even (PF=1).
0F 4A /r	CMOVPE r32, r/m32	RM	Valid	Valid	Move if parity even (PF=1).
REX.W + 0F 4A /r	CMOVPE r64, r/m64	RM	Valid	N.E.	Move if parity even (PF=1).

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 4B /r	CMOVPO r16, r/m16	RM	Valid	Valid	Move if parity odd (PF=0).
0F 4B /r	CMOVPO r32, r/m32	RM	Valid	Valid	Move if parity odd (PF=0).
REX.W + 0F 4B /r	CMOVPO r64, r/m64	RM	Valid	N.E.	Move if parity odd (PF=0).
0F 48 /r	CMOVS r16, r/m16	RM	Valid	Valid	Move if sign (SF=1).
0F 48 /r	CMOVS r32, r/m32	RM	Valid	Valid	Move if sign (SF=1).
REX.W + 0F 48 /r	CMOVS r64, r/m64	RM	Valid	N.E.	Move if sign (SF=1).
0F 44 /r	CMOVZ r16, r/m16	RM	Valid	Valid	Move if zero (ZF=1).
0F 44 /r	CMOVZ r32, r/m32	RM	Valid	Valid	Move if zero (ZF=1).
REX.W + 0F 44 /r	CMOVZ r64, r/m64	RM	Valid	N.E.	Move if zero (ZF=1).

Instruction Operand Encoding

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

Description

Each of the CMOVcc instructions performs a move operation if the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) are in a specified state (or condition). A condition code (*cc*) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, a move is not performed and execution continues with the instruction following the CMOV*cc* instruction.

Specifically, CMOVcc loads data from its source operand into a temporary register unconditionally (regardless of the condition code and the status flags in the EFLAGS register). If the condition code associated with the instruction (cc) is satisfied, the data in the temporary register is then copied into the instruction's destination operand.

These instructions can move 16-bit, 32-bit or 64-bit values from memory to a general-purpose register or from one general-purpose register to another. Conditional moves of 8-bit register operands are not supported.

The condition for each CMOV*cc* mnemonic is given in the description column of the above table. The terms “less” and “greater” are used for comparisons of signed integers and the terms “above” and “below” are used for unsigned integers.

Because a particular state of the status flags can sometimes be interpreted in two ways, two mnemonics are defined for some opcodes. For example, the CMOVA (conditional move if above) instruction and the CMOVNBE (conditional move if not below or equal) instruction are alternate mnemonics for the opcode 0F 47H.

The CMOV*cc* instructions were introduced in P6 family processors; however, these instructions may not be supported by all IA-32 processors. Software can determine if the CMOV*cc* instructions are supported by checking the processor’s feature information with the CPUID instruction (see “CPUID—CPU Identification” in this chapter).

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

Operation

```
temp := SRC

IF condition TRUE
    THEN DEST := temp;
ELSE IF (OperandSize = 32 and IA-32e mode active)
    THEN DEST[63:32] := 0;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

CMP—Compare Two Operands

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
3C ib	CMP AL, imm8	I	Valid	Valid	Compare imm8 with AL.
3D iw	CMP AX, imm16	I	Valid	Valid	Compare imm16 with AX.
3D id	CMP EAX, imm32	I	Valid	Valid	Compare imm32 with EAX.
REX.W + 3D id	CMP RAX, imm32	I	Valid	N.E.	Compare imm32 sign-extended to 64-bits with RAX.
80 /7 ib	CMP r/m8[1], imm8	MI	Valid	Valid	Compare imm8 with r/m8.
81 /7 iw	CMP r/m16, imm16	MI	Valid	Valid	Compare imm16 with r/m16.
81 /7 id	CMP r/m32, imm32	MI	Valid	Valid	Compare imm32 with r/m32.
REX.W + 81 /7 id	CMP r/m64, imm32	MI	Valid	N.E.	Compare imm32 sign-extended to 64-bits with r/m64.
83 /7 ib	CMP r/m16, imm8	MI	Valid	Valid	Compare imm8 with r/m16.
83 /7 ib	CMP r/m32, imm8	MI	Valid	Valid	Compare imm8 with r/m32.
REX.W + 83 /7 ib	CMP r/m64, imm8	MI	Valid	N.E.	Compare imm8 with r/m64.
38 /r	CMP r/m8[1], r8[1]	MR	Valid	Valid	Compare r8 with r/m8.
39 /r	CMP r/m16, r16	MR	Valid	Valid	Compare r16 with r/m16.
39 /r	CMP r/m32, r32	MR	Valid	Valid	Compare r32 with r/m32.
REX.W + 39 /r	CMP r/m64,r64	MR	Valid	N.E.	Compare r64 with r/m64.
3A /r	CMP r8[1], r/m8[1]	RM	Valid	Valid	Compare r/m8 with r8.
3B /r	CMP r16, r/m16	RM	Valid	Valid	Compare r/m16 with r16.
3B /r	CMP r32, r/m32	RM	Valid	Valid	Compare r/m32 with r32.
REX.W + 3B /r	CMP r64, r/m64	RM	Valid	N.E.	Compare r/m64 with r64.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	N/A	N/A
MR	ModRM:r/m (r)	ModRM:reg (r)	N/A	N/A
MI	ModRM:r/m (r)	imm8/16/32	N/A	N/A
I	AL/AX/EAX/RAX (r)	imm8/16/32	N/A	N/A

Description

Compares the first source operand with the second source operand and sets the status flags in the EFLAGS register according to the results. The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction. When an immediate value is used as an operand, it is sign-extended to the length of the first operand.

The condition codes used by the J*cc*, CMOV*cc*, and SET*cc* instructions are based on the results of a CMP instruction. Appendix B, "EFLAGS Condition Codes," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, shows the relationship of the status flags and the condition codes.

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

Operation

```
temp := SRC1 – SignExtend(SRC2);
ModifyStatusFlags; (* Modify status flags in the same manner as the SUB instruction*)
```

Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are set according to the result.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

CMP*cc*XADD—Compare and Add if Condition is Met

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 E6 !(11):rrr:bbb CMPBEXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If below or equal (CF=1 or ZF=1), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 E6 !(11):rrr:bbb CMPBEXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If below or equal (CF=1 or ZF=1), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.
VEX.128.66.0F38.W0 E2 !(11):rrr:bbb CMPBXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If below (CF=1), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 E2 !(11):rrr:bbb CMPBXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If below (CF=1), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.
VEX.128.66.0F38.W0 EE !(11):rrr:bbb CMPLEXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If less or equal (ZF=1 or SF≠OF), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 EE !(11):rrr:bbb CMPLEXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If less or equal (ZF=1 or SF≠OF), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.
VEX.128.66.0F38.W0 EC !(11):rrr:bbb CMPLXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If less (SF≠OF), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 EC !(11):rrr:bbb CMPLXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If less (SF≠OF), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.
VEX.128.66.0F38.W0 E7 !(11):rrr:bbb CMPNBEXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If not below or equal (CF=0 and ZF=0), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 E7 !(11):rrr:bbb CMPNBEXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If not below or equal (CF=0 and ZF=0), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.
VEX.128.66.0F38.W0 E3 !(11):rrr:bbb CMPNBXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If not below (CF=0), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 E3 !(11):rrr:bbb CMPNBXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If not below (CF=0), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.
VEX.128.66.0F38.W0 EF !(11):rrr:bbb CMPNLEXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If not less or equal (ZF=0 and SF=OF), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 EF !(11):rrr:bbb CMPNLEXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If not less or equal (ZF=0 and SF=OF), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.
VEX.128.66.0F38.W0 ED !(11):rrr:bbb CMPNLXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If not less (SF=OF), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 ED !(11):rrr:bbb CMPNLXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If not less (SF=OF), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.
VEX.128.66.0F38.W0 E1 !(11):rrr:bbb CMPNOXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If not overflow (OF=0), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 E1 !(11):rrr:bbb CMPNOXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If not overflow (OF=0), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 EB !(11):rrr:bbb CMPNPXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If not parity (PF=0), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 EB !(11):rrr:bbb CMPNPXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If not parity (PF=0), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.
VEX.128.66.0F38.W0 E9 !(11):rrr:bbb CMPNSXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If not sign (SF=0), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 E9 !(11):rrr:bbb CMPNSXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If not sign (SF=0), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.
VEX.128.66.0F38.W0 E5 !(11):rrr:bbb CMPNZXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If not zero (ZF=0), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 E5 !(11):rrr:bbb CMPNZXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If not zero (ZF=0), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.
VEX.128.66.0F38.W0 E0 !(11):rrr:bbb CMPOXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If overflow (OF=1), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 E0 !(11):rrr:bbb CMPOXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If overflow (OF=1), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.
VEX.128.66.0F38.W0 EA !(11):rrr:bbb CMPPXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If parity (PF=1), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 EA !(11):rrr:bbb CMPPXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If parity (PF=1), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 E8 !(11):rrr:bbb CMPSXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If sign (SF=1), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 E8 !(11):rrr:bbb CMPSXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If sign (SF=1), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.
VEX.128.66.0F38.W0 E4 !(11):rrr:bbb CMPZXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If zero (ZF=1), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 E4 !(11):rrr:bbb CMPZXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If zero (ZF=1), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.

Instruction Operand Encoding[1]

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

Description

This instruction compares the value from memory with the value of the second operand. If the specified condition is met, then the processor will add the third operand to the memory operand and write it into memory, else the memory is unchanged by this instruction.

This instruction must have MODRM.MOD equal to 0, 1, or 2. The value 3 for MODRM.MOD is reserved and will cause an invalid opcode exception (#UD).

The second operand is always updated with the original value of the memory operand. The EFLAGS conditions are updated from the results of the comparison.The instruction uses an implicit lock. This instruction does not permit the use of an explicit lock prefix.

Operation

CMPCCXADD srcdest1, srcdest2, src3

```
tmp1 := load lock srcdest1
tmp2 := tmp1 + src3
EFLAGS.CS,OF,SF,ZF,AF,PF := CMP tmp1, srcdest2
IF <condition>:
    srcdest1 := store unlock tmp2
ELSE
    srcdest1 := store unlock tmp1
srcdest2 :=tmp1
```

1. ModRM.MOD != 011B

Flags Affected

The EFLAGS conditions are updated from the results of the comparison.

Intel C/C++ Compiler Intrinsic Equivalent

CMPCCXADD int _cmpccxadd_epi32 (void* __A, int __B, int __C, const int __D);
CMPCCXADD __int64 _cmpccxadd_epi64 (void* __A, __int64 __B, __int64 __C, const int __D);

SIMD Floating-Point Exceptions

None.

Exceptions

Exceptions Type 14; see Table 2-31.

CMPPD—Compare Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F C2 /r ib CMPPD xmm1, xmm2/m128, imm8	A	V/V	SSE2	Compare packed double precision floating-point values in xmm2/m128 and xmm1 using bits 2:0 of imm8 as a comparison predicate.
VEX.128.66.0F.WIG C2 /r ib VCMPPD xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Compare packed double precision floating-point values in xmm3/m128 and xmm2 using bits 4:0 of imm8 as a comparison predicate.
VEX.256.66.0F.WIG C2 /r ib VCMPPD ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Compare packed double precision floating-point values in ymm3/m256 and ymm2 using bits 4:0 of imm8 as a comparison predicate.
EVEX.128.66.0F.W1 C2 /r ib VCMPPD k1 {k2}, xmm2, xmm3/m128/m64bcst, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed double precision floating-point values in xmm3/m128/m64bcst and xmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.256.66.0F.W1 C2 /r ib VCMPPD k1 {k2}, ymm2, ymm3/m256/m64bcst, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed double precision floating-point values in ymm3/m256/m64bcst and ymm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.512.66.0F.W1 C2 /r ib VCMPPD k1 {k2}, zmm2, zmm3/m512/m64bcst {sae}, imm8	C	V/V	AVX512F OR AVX10.1	Compare packed double precision floating-point values in zmm3/m512/m64bcst and zmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Performs a SIMD compare of the packed double precision floating-point values in the second source operand and the first source operand and returns the result of the comparison to the destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands.

EVEX encoded versions: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is an opmask register. Comparison results are written to the destination operand under the writemask k2. Each comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false).

VEX.256 encoded version: The first source operand (second operand) is a YMM register. The second source operand (third operand) can be a YMM register or a 256-bit memory location. The destination operand (first operand) is a YMM register. Four comparisons are performed with results written to the destination operand. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged. Two comparisons are performed with results

written to bits 127:0 of the destination operand. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. Two comparisons are performed with results written to bits 127:0 of the destination operand.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX or EVEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-8). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-8). Bits 3 through 7 of the immediate are reserved.

Table 3-8. Comparison Predicate for CMPPD and CMPPS Instructions

Predicate	imm8 Value	Description	Result: A Is 1st Operand, B Is 2nd Operand				Signals #IA on QNAN
			A >B	A < B	A = B	Unordered[1]	
EQ_OQ (EQ)	0H	Equal (ordered, non-signaling)	False	False	True	False	No
LT_OS (LT)	1H	Less-than (ordered, signaling)	False	True	False	False	Yes
LE_OS (LE)	2H	Less-than-or-equal (ordered, signaling)	False	True	True	False	Yes
UNORD_Q (UNORD)	3H	Unordered (non-signaling)	False	False	False	True	No
NEQ_UQ (NEQ)	4H	Not-equal (unordered, non-signaling)	True	True	False	True	No
NLT_US (NLT)	5H	Not-less-than (unordered, signaling)	True	False	True	True	Yes
NLE_US (NLE)	6H	Not-less-than-or-equal (unordered, signaling)	True	False	False	True	Yes
ORD_Q (ORD)	7H	Ordered (non-signaling)	True	True	True	False	No
EQ_UQ	8H	Equal (unordered, non-signaling)	False	False	True	True	No
NGE_US (NGE)	9H	Not-greater-than-or-equal (unordered, signaling)	False	True	False	True	Yes
NGT_US (NGT)	AH	Not-greater-than (unordered, signaling)	False	True	True	True	Yes
FALSE_OQ(FALSE)	BH	False (ordered, non-signaling)	False	False	False	False	No
NEQ_OQ	CH	Not-equal (ordered, non-signaling)	True	True	False	False	No
GE_OS (GE)	DH	Greater-than-or-equal (ordered, signaling)	True	False	True	False	Yes
GT_OS (GT)	EH	Greater-than (ordered, signaling)	True	False	False	False	Yes
TRUE_UQ(TRUE)	FH	True (unordered, non-signaling)	True	True	True	True	No
EQ_OS	10H	Equal (ordered, signaling)	False	False	True	False	Yes
LT_OQ	11H	Less-than (ordered, nonsignaling)	False	True	False	False	No
LE_OQ	12H	Less-than-or-equal (ordered, nonsignaling)	False	True	True	False	No
UNORD_S	13H	Unordered (signaling)	False	False	False	True	Yes
NEQ_US	14H	Not-equal (unordered, signaling)	True	True	False	True	Yes
NLT_UQ	15H	Not-less-than (unordered, nonsignaling)	True	False	True	True	No
NLE_UQ	16H	Not-less-than-or-equal (unordered, nonsignaling)	True	False	False	True	No
ORD_S	17H	Ordered (signaling)	True	True	True	False	Yes
EQ_US	18H	Equal (unordered, signaling)	False	False	True	True	Yes

Table 3-8. Comparison Predicate for CMPPD and CMPPS Instructions (Contd.)

Predicate	imm8 Value	Description	Result: A Is 1st Operand, B Is 2nd Operand				Signals #IA on QNAN
			A >B	A < B	A = B	Unordered[1]	
NGE_UQ	19H	Not-greater-than-or-equal (unordered, non-signaling)	False	True	False	True	No
NGT_UQ	1AH	Not-greater-than (unordered, nonsignaling)	False	True	True	True	No
FALSE_OS	1BH	False (ordered, signaling)	False	False	False	False	Yes
NEQ_OS	1CH	Not-equal (ordered, signaling)	True	True	False	False	Yes
GE_OQ	1DH	Greater-than-or-equal (ordered, nonsignal-ing)	True	False	True	False	No
GT_OQ	1EH	Greater-than (ordered, nonsignaling)	True	False	False	False	No
TRUE_US	1FH	True (unordered, signaling)	True	True	True	True	Yes

NOTES:

1. If either operand A or B is a NAN.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.01H:ECX.AVX[28] = 0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal" relations predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPD instruction, for processors with "CPUID.01H:ECX.AVX[28] = 0". See Table 3-9. The compiler should treat reserved imm8 values as illegal syntax.

Table 3-9. Pseudo-Op and CMPPD Implementation

Pseudo-Op	CMPPD Implementation
CMPEQPD xmm1, xmm2	CMPPD xmm1, xmm2, 0
CMPLTPD xmm1, xmm2	CMPPD xmm1, xmm2, 1
CMPLEPD xmm1, xmm2	CMPPD xmm1, xmm2, 2
CMPUNORDPD xmm1, xmm2	CMPPD xmm1, xmm2, 3
CMPNEQPD xmm1, xmm2	CMPPD xmm1, xmm2, 4
CMPNLTPD xmm1, xmm2	CMPPD xmm1, xmm2, 5
CMPNLEPD xmm1, xmm2	CMPPD xmm1, xmm2, 6
CMPORDPD xmm1, xmm2	CMPPD xmm1, xmm2, 7

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with "CPUID.01H:ECX.AVX[28] = 1" implement the full complement of 32 predicates shown in Table 3-10, software emulation is no longer needed. Compilers and assemblers may implement the following

three-operand pseudo-ops in addition to the four-operand VCMPPD instruction. See Table 3-10, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. The compiler should treat reserved imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPPD instructions in a similar fashion by extending the syntax listed in Table 3-10.

Table 3-10. Pseudo-Op and VCMPPD Implementation

Pseudo-Op	CMPPD Implementation
VCMPEQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 0*
VCMPLTPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 1*
VCMPLEPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 2*
VCMPUNORDPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 3*
VCMPNEQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 4*
VCMPNLTPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 5*
VCMPNLEPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 6*
VCMPORDPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 7*
VCMPEQ_UQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 8*
VCMPNGEPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 9*
VCMPNGTPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 0AH*
VCMPFALSEPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 0BH*
VCMPNEQ_OQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 0CH*
VCMPGEPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 0DH*
VCMPGTPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 0EH*
VCMPTRUEPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 0FH*
VCMPEQ_OSPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 10H*
VCMPLT_OQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 11H*
VCMPLE_OQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 12H*
VCMPUNORD_SPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 13H*
VCMPNEQ_USPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 14H*
VCMPNLT_UQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 15H*
VCMPNLE_UQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 16H*
VCMPORD_SPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 17H*
VCMPEQ_USPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 18H*
VCMPNGE_UQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 19H*
VCMPNGT_UQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 1AH*
VCMPFALSE_OSPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 1BH*
VCMPNEQ_OSPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 1CH*
VCMPGE_OQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 1DH*
VCMPGT_OQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 1EH*
VCMPTRUE_USPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 1FH*

Operation

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

VCMPPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    CMP := SRC1[i+63:i] OP5 SRC2[63:0]
                ELSE
                    CMP := SRC1[i+63:i] OP5 SRC2[i+63:i]
            FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0                ; zeroing-masking only
    FI;
```

```
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VCMPPD (VEX.256 Encoded Version)

```
CMP0 := SRC1[63:0] OP5 SRC2[63:0];
CMP1 := SRC1[127:64] OP5 SRC2[127:64];
CMP2 := SRC1[191:128] OP5 SRC2[191:128];
CMP3 := SRC1[255:192] OP5 SRC2[255:192];
IF CMP0 = TRUE
    THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] := 0000000000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[127:64] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] := 0000000000000000H; FI;
IF CMP2 = TRUE
    THEN DEST[191:128] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[191:128] := 0000000000000000H; FI;
IF CMP3 = TRUE
    THEN DEST[255:192] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[255:192] := 0000000000000000H; FI;
DEST[MAXVL-1:256] := 0
```

VCMPPD (VEX.128 Encoded Version)

```
CMP0 := SRC1[63:0] OP5 SRC2[63:0];
CMP1 := SRC1[127:64] OP5 SRC2[127:64];
IF CMP0 = TRUE
    THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] := 0000000000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[127:64] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] := 0000000000000000H; FI;
DEST[MAXVL-1:128] := 0
```

CMPPD (128-bit Legacy SSE Version)

```
CMP0 := SRC1[63:0] OP3 SRC2[63:0];
CMP1 := SRC1[127:64] OP3 SRC2[127:64];
IF CMP0 = TRUE
    THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] := 0000000000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[127:64] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] := 0000000000000000H; FI;
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCMPPD __mmask8 _mm512_cmp_pd_mask(__m512d a, __m512d b, int imm);
VCMPPD __mmask8 _mm512_cmp_round_pd_mask(__m512d a, __m512d b, int imm, int sae);
VCMPPD __mmask8 _mm512_mask_cmp_pd_mask(__mmask8 k1, __m512d a, __m512d b, int imm);
VCMPPD __mmask8 _mm512_mask_cmp_round_pd_mask(__mmask8 k1, __m512d a, __m512d b, int imm, int sae);
VCMPPD __mmask8 _mm256_cmp_pd_mask(__m256d a, __m256d b, int imm);
VCMPPD __mmask8 _mm256_mask_cmp_pd_mask(__mmask8 k1, __m256d a, __m256d b, int imm);
VCMPPD __mmask8 _mm_cmp_pd_mask(__m128d a, __m128d b, int imm);
VCMPPD __mmask8 _mm_mask_cmp_pd_mask(__mmask8 k1, __m128d a, __m128d b, int imm);
VCMPPD __m256 _mm256_cmp_pd(__m256d a, __m256d b, int imm)

(V)CMPPD __m128 _mm_cmp_pd(__m128d a, __m128d b, int imm)

SIMD Floating-Point Exceptions

Invalid if SNaN operand and invalid if QNaN and predicate as listed in Table 3-8, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, "Type 2 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

CMPPS—Compare Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C2 /r ib CMPPS xmm1, xmm2/m128, imm8	A	V/V	SSE	Compare packed single precision floating-point values in xmm2/m128 and xmm1 using bits 2:0 of imm8 as a comparison predicate.
VEX.128.0F.WIG C2 /r ib VCMPPS xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Compare packed single precision floating-point values in xmm3/m128 and xmm2 using bits 4:0 of imm8 as a comparison predicate.
VEX.256.0F.WIG C2 /r ib VCMPPS ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Compare packed single precision floating-point values in ymm3/m256 and ymm2 using bits 4:0 of imm8 as a comparison predicate.
EVEX.128.0F.W0 C2 /r ib VCMPPS k1 {k2}, xmm2, xmm3/m128/m32bcst, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed single precision floating-point values in xmm3/m128/m32bcst and xmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.256.0F.W0 C2 /r ib VCMPPS k1 {k2}, ymm2, ymm3/m256/m32bcst, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed single precision floating-point values in ymm3/m256/m32bcst and ymm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.512.0F.W0 C2 /r ib VCMPPS k1 {k2}, zmm2, zmm3/m512/m32bcst {sae}, imm8	C	V/V	AVX512F OR AVX10.1	Compare packed single precision floating-point values in zmm3/m512/m32bcst and zmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Performs a SIMD compare of the packed single precision floating-point values in the second source operand and the first source operand and returns the result of the comparison to the destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each of the pairs of packed values.

EVEX encoded versions: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is an opmask register. Comparison results are written to the destination operand under the writemask k2. Each comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false).

VEX.256 encoded version: The first source operand (second operand) is a YMM register. The second source operand (third operand) can be a YMM register or a 256-bit memory location. The destination operand (first operand) is a YMM register. Eight comparisons are performed with results written to the destination operand. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged. Four comparisons are performed with results written to bits 127:0 of the destination operand. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. Four comparisons are performed with results written to bits 127:0 of the destination operand.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix and EVEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-8). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-8). Bits 3 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.01H:ECX.AVX[28] = 0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal" relations predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPS instruction, for processors with "CPUID.01H:ECX.AVX[28] = 0". See Table 3-11. The compiler should treat reserved imm8 values as illegal syntax.

Table 3-11. Pseudo-Op and CMPPS Implementation

Pseudo-Op	CMPPS Implementation
CMPEQPS xmm1, xmm2	CMPPS xmm1, xmm2, 0
CMPLTPS xmm1, xmm2	CMPPS xmm1, xmm2, 1
CMPLEPS xmm1, xmm2	CMPPS xmm1, xmm2, 2
CMPUNORDPS xmm1, xmm2	CMPPS xmm1, xmm2, 3
CMPNEQPS xmm1, xmm2	CMPPS xmm1, xmm2, 4
CMPNLTPS xmm1, xmm2	CMPPS xmm1, xmm2, 5
CMPNLEPS xmm1, xmm2	CMPPS xmm1, xmm2, 6
CMPORDPS xmm1, xmm2	CMPPS xmm1, xmm2, 7

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with "CPUID.01H:ECX.AVX[28] = 1" implement the full complement of 32 predicates shown in Table 3-12, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPPS instruction. See Table 3-12, where the notation of reg1 and reg2 represent either XMM registers or YMM registers. The compiler should treat reserved imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPPS instructions in a similar fashion by extending the syntax listed in Table 3-12.

Table 3-12. Pseudo-Op and VCMPPS Implementation

Pseudo-Op	CMPPS Implementation
VCMPEQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0*
VCMPLTPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1*
VCMPLEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 2*
VCMPUNORDPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 3*
VCMPNEQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 4*
VCMPNLTPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 5*
VCMPNLEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 6*
VCMPORDPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 7*
VCMPEQ_UQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 8*
VCMPNGEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 9*
VCMPNGTPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0AH*
VCMPFALSEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0BH*
VCMPNEQ_OQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0CH*
VCMPGEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0DH*
VCMPGTPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0EH*
VCMPTRUEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0FH*
VCMPEQ_OSPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 10H*
VCMPLT_OQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 11H*
VCMPLE_OQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 12H*
VCMPUNORD_SPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 13H*
VCMPNEQ_USPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 14H*
VCMPNLT_UQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 15H*
VCMPNLE_UQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 16H*
VCMPORD_SPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 17H*
VCMPEQ_USPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 18H*
VCMPNGE_UQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 19H*
VCMPNGT_UQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1AH*
VCMPFALSE_OSPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1BH*
VCMPNEQ_OSPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1CH*
VCMPGE_OQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1DH*
VCMPGT_OQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1EH*
VCMPTRUE_USPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1FH*

Operation

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

VCMPPS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    CMP := SRC1[i+31:i] OP5 SRC2[31:0]
                ELSE
                    CMP := SRC1[i+31:i] OP5 SRC2[i+31:i]
            FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0                    ; zeroing-masking onlyFI;
    FI;
```

```
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VCMPPS (VEX.256 Encoded Version)

```
CMP0 := SRC1[31:0] OP5 SRC2[31:0];
CMP1 := SRC1[63:32] OP5 SRC2[63:32];
CMP2 := SRC1[95:64] OP5 SRC2[95:64];
CMP3 := SRC1[127:96] OP5 SRC2[127:96];
CMP4 := SRC1[159:128] OP5 SRC2[159:128];
CMP5 := SRC1[191:160] OP5 SRC2[191:160];
CMP6 := SRC1[223:192] OP5 SRC2[223:192];
CMP7 := SRC1[255:224] OP5 SRC2[255:224];
IF CMP0 = TRUE
    THEN DEST[31:0] :=FFFFFFFFH;
    ELSE DEST[31:0] := 000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[63:32] := FFFFFFFFH;
    ELSE DEST[63:32] :=000000000H; FI;
IF CMP2 = TRUE
    THEN DEST[95:64] := FFFFFFFFH;
    ELSE DEST[95:64] := 000000000H; FI;
IF CMP3 = TRUE
    THEN DEST[127:96] := FFFFFFFFH;
    ELSE DEST[127:96] := 000000000H; FI;
IF CMP4 = TRUE
    THEN DEST[159:128] := FFFFFFFFH;
    ELSE DEST[159:128] := 000000000H; FI;
IF CMP5 = TRUE
    THEN DEST[191:160] := FFFFFFFFH;
    ELSE DEST[191:160] := 000000000H; FI;
IF CMP6 = TRUE
    THEN DEST[223:192] := FFFFFFFFH;
    ELSE DEST[223:192] :=000000000H; FI;
IF CMP7 = TRUE
    THEN DEST[255:224] := FFFFFFFFH;
    ELSE DEST[255:224] := 000000000H; FI;
DEST[MAXVL-1:256] := 0
```

VCMPPS (VEX.128 Encoded Version)

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

```
    ELSE DEST[127:96] :=000000000H; FI;
DEST[MAXVL-1:128] := 0
```

CMPPS (128-bit Legacy SSE Version)

```
CMP0 := SRC1[31:0] OP3 SRC2[31:0];
CMP1 := SRC1[63:32] OP3 SRC2[63:32];
CMP2 := SRC1[95:64] OP3 SRC2[95:64];
CMP3 := SRC1[127:96] OP3 SRC2[127:96];
IF CMP0 = TRUE
    THEN DEST[31:0] :=FFFFFFFFH;
    ELSE DEST[31:0] := 000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[63:32] := FFFFFFFFH;
    ELSE DEST[63:32] := 000000000H; FI;
IF CMP2 = TRUE
    THEN DEST[95:64] := FFFFFFFFH;
    ELSE DEST[95:64] := 000000000H; FI;
IF CMP3 = TRUE
    THEN DEST[127:96] := FFFFFFFFH;
    ELSE DEST[127:96] :=000000000H; FI;
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCMPPS __mmask16 _mm512_cmp_ps_mask(__m512 a, __m512 b, int imm);
VCMPPS __mmask16 _mm512_cmp_round_ps_mask(__m512 a, __m512 b, int imm, int sae);
VCMPPS __mmask16 _mm512_mask_cmp_ps_mask(__mmask16 k1, __m512 a, __m512 b, int imm);
VCMPPS __mmask16 _mm512_mask_cmp_round_ps_mask(__mmask16 k1, __m512 a, __m512 b, int imm, int sae);
VCMPPS __mmask8 _mm256_cmp_ps_mask(__m256 a, __m256 b, int imm);
VCMPPS __mmask8 _mm256_mask_cmp_ps_mask(__mmask8 k1, __m256 a, __m256 b, int imm);
VCMPPS __mmask8 _mm_cmp_ps_mask(__m128 a, __m128 b, int imm);
VCMPPS __mmask8 _mm_mask_cmp_ps_mask(__mmask8 k1, __m128 a, __m128 b, int imm);
VCMPPS __m256 _mm256_cmp_ps(__m256 a, __m256 b, int imm)
CMPPS __m128 _mm_cmp_ps(__m128 a, __m128 b, int imm)

SIMD Floating-Point Exceptions

Invalid if SNaN operand and invalid if QNaN and predicate as listed in Table 3-8, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

CMPS/CMPSB/CMPSW/CMPSD/CMPSQ—Compare String Operands

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
A6	CMPS *m8, m8*	ZO	Valid	Valid	For legacy mode, compare byte at address DS:(E)SI with byte at address ES:(E)DI; For 64-bit mode compare byte at address (R|E)SI to byte at address (R|E)DI. The status flags are set accordingly.
A7	CMPS *m16, m16*	ZO	Valid	Valid	For legacy mode, compare word at address DS:(E)SI with word at address ES:(E)DI; For 64-bit mode compare word at address (R|E)SI with word at address (R|E)DI. The status flags are set accordingly.
A7	CMPS *m32, m32*	ZO	Valid	Valid	For legacy mode, compare dword at address DS:(E)SI at dword at address ES:(E)DI; For 64-bit mode compare dword at address (R|E)SI at dword at address (R|E)DI. The status flags are set accordingly.
REX.W + A7	CMPS *m64, m64*	ZO	Valid	N.E.	Compares quadword at address (R|E)SI with quadword at address (R|E)DI and sets the status flags accordingly.
A6	CMPSB	ZO	Valid	Valid	For legacy mode, compare byte at address DS:(E)SI with byte at address ES:(E)DI; For 64-bit mode compare byte at address (R|E)SI with byte at address (R|E)DI. The status flags are set accordingly.
A7	CMPSW	ZO	Valid	Valid	For legacy mode, compare word at address DS:(E)SI with word at address ES:(E)DI; For 64-bit mode compare word at address (R|E)SI with word at address (R|E)DI. The status flags are set accordingly.
A7	CMPSD	ZO	Valid	Valid	For legacy mode, compare dword at address DS:(E)SI with dword at address ES:(E)DI; For 64-bit mode compare dword at address (R|E)SI with dword at address (R|E)DI. The status flags are set accordingly.
REX.W + A7	CMPSQ	ZO	Valid	N.E.	Compares quadword at address (R|E)SI with quadword at address (R|E)DI and sets the status flags accordingly.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Compares the byte, word, doubleword, or quadword specified with the first source operand with the byte, word, doubleword, or quadword specified with the second source operand and sets the status flags in the EFLAGS register according to the results.

Both source operands are located in memory. The address of the first source operand is read from DS:SI, DS:ESI or RSI (depending on the address-size attribute of the instruction is 16, 32, or 64, respectively). The address of the second source operand is read from ES:DI, ES:EDI or RDI (again depending on the address-size attribute of the instruction is 16, 32, or 64). The DS segment may be overridden with a segment override prefix, but the ES segment cannot be overridden.

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operands" form and the "no-operands" form. The explicit-operands form (specified with the CMPS mnemonic) allows the two source operands to be specified explicitly. Here, the source operands should be symbols that indicate the size and location of the source values. This explicit-operand form is provided to allow documentation. However, note that the documentation provided by this form can be misleading. That is, the source operand symbols must specify the correct type (size) of the operands (bytes, words, or doublewords, quadwords), but they do not have to specify the correct loca-

tion. Locations of the source operands are always specified by the DS:(E)SI (or RSI) and ES:(E)DI (or RDI) registers, which must be loaded correctly before the compare string instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the CMPS instructions. Here also the DS:(E)SI (or RSI) and ES:(E)DI (or RDI) registers are assumed by the processor to specify the location of the source operands. The size of the source operands is selected with the mnemonic: CMPSB (byte comparison), CMPSW (word comparison), CMPSD (doubleword comparison), or CMPSQ (quadword comparison using REX.W).

After the comparison, the (E/R)SI and (E/R)DI registers increment or decrement automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E/R)SI and (E/R)DI register increment; if the DF flag is 1, the registers decrement.) The registers increment or decrement by 1 for byte operations, by 2 for word operations, 4 for doubleword operations. If operand size is 64, RSI and RDI registers increment by 8 for quadword operations.

The CMPS, CMPSB, CMPSW, CMPSD, and CMPSQ instructions can be preceded by the REPE prefix or the REPNE prefix for block comparisons. See "REPE/REPZ—Repeat String Operation While Zero (Prefix)" and "REPNE/REPNZ—Repeat String Operation While Not Zero (Prefix)" in Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B, for a description of the REPE and REPNE prefixes.

In 64-bit mode, the instruction's default address size is 64 bits, 32 bit address size is supported using the prefix 67H. Use of the REX.W prefix promotes doubleword operation to 64 bits (see CMPSQ). See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
temp := SRC1 - SRC2;
SetStatusFlags(temp);

IF (64-Bit Mode)
    THEN
        IF (Byte comparison)
        THEN IF DF = 0
            THEN
                (R|E)SI := (R|E)SI + 1;
                (R|E)DI := (R|E)DI + 1;
            ELSE
                (R|E)SI := (R|E)SI – 1;
                (R|E)DI := (R|E)DI – 1;
            FI;
        ELSE IF (Word comparison)
            THEN IF DF = 0
                THEN
                    (R|E)SI := (R|E)SI + 2;
                    (R|E)DI := (R|E)DI + 2;
                ELSE
                    (R|E)SI := (R|E)SI – 2;
                    (R|E)DI := (R|E)DI – 2;
                FI;
        ELSE IF (Doubleword comparison)
            THEN IF DF = 0
                THEN
                    (R|E)SI := (R|E)SI + 4;
                    (R|E)DI := (R|E)DI + 4;
                ELSE
                    (R|E)SI := (R|E)SI – 4;
                    (R|E)DI := (R|E)DI – 4;
                FI;
        ELSE (* Quadword comparison *)
```

```
            THEN IF DF = 0
                (R|E)SI := (R|E)SI + 8;
                (R|E)DI := (R|E)DI + 8;
            ELSE
                (R|E)SI := (R|E)SI – 8;
                (R|E)DI := (R|E)DI – 8;
            FI;
        FI;
    ELSE (* Non-64-bit Mode *)
        IF (byte comparison)
        THEN IF DF = 0
            THEN
                (E)SI := (E)SI + 1;
                (E)DI := (E)DI + 1;
            ELSE
                (E)SI := (E)SI – 1;
                (E)DI := (E)DI – 1;
            FI;
        ELSE IF (Word comparison)
            THEN IF DF = 0
                (E)SI := (E)SI + 2;
                (E)DI := (E)DI + 2;
            ELSE
                (E)SI := (E)SI – 2;
                (E)DI := (E)DI – 2;
            FI;
        ELSE (* Doubleword comparison *)
            THEN IF DF = 0
                (E)SI := (E)SI + 4;
                (E)DI := (E)DI + 4;
            ELSE
                (E)SI := (E)SI – 4;
                (E)DI := (E)DI – 4;
            FI;
        FI;
FI;
```

Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are set according to the temporary result of the comparison.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

CMPSD—Compare Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F C2 /r ib CMPSD xmm1, xmm2/m64, imm8	A	V/V	SSE2	Compare low double precision floating-point value in xmm2/m64 and xmm1 using bits 2:0 of imm8 as comparison predicate.
VEX.LIG.F2.0F.WIG C2 /r ib VCMPSD xmm1, xmm2, xmm3/m64, imm8	B	V/V	AVX	Compare low double precision floating-point value in xmm3/m64 and xmm2 using bits 4:0 of imm8 as comparison predicate.
EVEX.LLIG.F2.0F.W1 C2 /r ib VCMPSD k1 {k2}, xmm2, xmm3/m64{sae}, imm8	C	V/V	AVX512F OR AVX10.1	Compare low double precision floating-point value in xmm3/m64 and xmm2 using bits 4:0 of imm8 as comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Compares the low double precision floating-point values in the second source operand and the first source operand and returns the result of the comparison to the destination operand. The comparison predicate operand (immediate operand) specifies the type of comparison performed.

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 64-bit memory location. Bits (MAXVL-1:64) of the corresponding YMM destination register remain unchanged. The comparison result is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 64-bit memory location. The result is stored in the low quadword of the destination operand; the high quadword is filled with the contents of the high quadword of the first source operand. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. The comparison result is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

EVEX encoded version: The first source operand (second operand) is an XMM register. The second source operand can be a XMM register or a 64-bit memory location. The destination operand (first operand) is an opmask register. The comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false), written to the destination starting from the LSB according to the writemask k2. Bits (MAX_KL-1:128) of the destination register are cleared.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-8). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-8). Bits 3 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.01H:ECX.AVX[28] = 0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made

either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSD instruction, for processors with "CPUID.01H:ECX.AVX[28] = 0". See Table 3-13. The compiler should treat reserved imm8 values as illegal syntax.

Table 3-13. Pseudo-Op and CMPSD Implementation

Pseudo-Op	CMPSD Implementation
CMPEQSD xmm1, xmm2	CMPSD xmm1, xmm2, 0
CMPLTSD xmm1, xmm2	CMPSD xmm1, xmm2, 1
CMPLESD xmm1, xmm2	CMPSD xmm1, xmm2, 2
CMPUNORDSD xmm1, xmm2	CMPSD xmm1, xmm2, 3
CMPNEQSD xmm1, xmm2	CMPSD xmm1, xmm2, 4
CMPNLTSD xmm1, xmm2	CMPSD xmm1, xmm2, 5
CMPNLESD xmm1, xmm2	CMPSD xmm1, xmm2, 6
CMPORDSD xmm1, xmm2	CMPSD xmm1, xmm2, 7

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with "CPUID.01H:ECX.AVX[28] = 1" implement the full complement of 32 predicates shown in Table 3-14, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPSD instruction. See Table 3-14, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. The compiler should treat reserved imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPSD instructions in a similar fashion by extending the syntax listed in Table 3-14.

Table 3-14. Pseudo-Op and VCMPSD Implementation

Pseudo-Op	CMPSD Implementation
VCMPEQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0
VCMPLTSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1
VCMPLESD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 2
VCMPUNORDSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 3
VCMPNEQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 4
VCMPNLTSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 5
VCMPNLESD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 6
VCMPORDSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 7
VCMPEQ_UQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 8
VCMPNGESD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 9
VCMPNGTSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0AH
VCMPFALSESD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0BH
VCMPNEQ_OQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0CH
VCMPGESD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0DH

Table 3-14. Pseudo-Op and VCMPSD Implementation (Contd.)

Pseudo-Op	CMPSD Implementation
VCMPGTSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0EH
VCMPTRUESD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0FH
VCMPEQ_OSSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 10H
VCMPLT_OQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 11H
VCMPLE_OQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 12H
VCMPUNORD_SSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 13H
VCMPNEQ_USSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 14H
VCMPNLT_UQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 15H
VCMPNLE_UQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 16H
VCMPORD_SSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 17H
VCMPEQ_USSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 18H
VCMPNGE_UQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 19H
VCMPNGT_UQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1AH
VCMPFALSE_OSSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1BH
VCMPNEQ_OSSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1CH
VCMPGE_OQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1DH
VCMPGT_OQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1EH
VCMPTRUE_USSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1FH

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

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP3 := EQ_OQ; OP5 := EQ_OQ;
    1: OP3 := LT_OS; OP5 := LT_OS;
    2: OP3 := LE_OS; OP5 := LE_OS;
    3: OP3 := UNORD_Q; OP5 := UNORD_Q;
    4: OP3 := NEQ_UQ; OP5 := NEQ_UQ;
    5: OP3 := NLT_US; OP5 := NLT_US;
    6: OP3 := NLE_US; OP5 := NLE_US;
    7: OP3 := ORD_Q; OP5 := ORD_Q;
    8: OP5 := EQ_UQ;
    9: OP5 := NGE_US;
    10: OP5 := NGT_US;
    11: OP5 := FALSE_OQ;
    12: OP5 := NEQ_OQ;
    13: OP5 := GE_OS;
    14: OP5 := GT_OS;
    15: OP5 := TRUE_UQ;
    16: OP5 := EQ_OS;
    17: OP5 := LT_OQ;
    18: OP5 := LE_OQ;
    19: OP5 := UNORD_S;
    20: OP5 := NEQ_US;
    21: OP5 := NLT_UQ;
```

```
    22: OP5 := NLE_UQ;
    23: OP5 := ORD_S;
    24: OP5 := EQ_US;
    25: OP5 := NGE_UQ;
    26: OP5 := NGT_UQ;
    27: OP5 := FALSE_OS;
    28: OP5 := NEQ_OS;
    29: OP5 := GE_OQ;
    30: OP5 := GT_OQ;
    31: OP5 := TRUE_US;
    DEFAULT: Reserved
ESAC;
```

VCMPSD (EVEX Encoded Version)

```
CMP0 := SRC1[63:0] OP5 SRC2[63:0];

IF k2[0] or *no writemask*
    THEN    IF CMP0 = TRUE
                THEN DEST[0] := 1;
                ELSE DEST[0] := 0; FI;
    ELSE    DEST[0] := 0                    ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] := 0
```

CMPSD (128-bit Legacy SSE Version)

```
CMP0 := DEST[63:0] OP3 SRC[63:0];
IF CMP0 = TRUE
THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] := 0000000000000000H; FI;
DEST[MAXVL-1:64] (Unmodified)
```

VCMPSD (VEX.128 Encoded Version)

```
CMP0 := SRC1[63:0] OP5 SRC2[63:0];
IF CMP0 = TRUE
THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] := 0000000000000000H; FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCMPSD __mmask8 _mm_cmp_sd_mask(__m128d a, __m128d b, int imm);
VCMPSD __mmask8 _mm_cmp_round_sd_mask(__m128d a, __m128d b, int imm, int sae);
VCMPSD __mmask8 _mm_mask_cmp_sd_mask(__mmask8 k1, __m128d a, __m128d b, int imm);
VCMPSD __mmask8 _mm_mask_cmp_round_sd_mask(__mmask8 k1, __m128d a, __m128d b, int imm, int sae);
(V)CMPSD __m128d _mm_cmp_sd(__m128d a, __m128d b, const int imm)

SIMD Floating-Point Exceptions

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in Table 3-8, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-49, "Type E3 Class Exception Conditions."

CMPSS—Compare Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F C2 /r ib CMPSS xmm1, xmm2/m32, imm8	A	V/V	SSE	Compare low single precision floating-point value in xmm2/m32 and xmm1 using bits 2:0 of imm8 as comparison predicate.
VEX.LIG.F3.0F.WIG C2 /r ib VCMPSS xmm1, xmm2, xmm3/m32, imm8	B	V/V	AVX	Compare low single precision floating-point value in xmm3/m32 and xmm2 using bits 4:0 of imm8 as comparison predicate.
EVEX.LLIG.F3.0F.W0 C2 /r ib VCMPSS k1 {k2}, xmm2, xmm3/m32{sae}, imm8	C	V/V	AVX512F OR AVX10.1	Compare low single precision floating-point value in xmm3/m32 and xmm2 using bits 4:0 of imm8 as comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Compares the low single precision floating-point values in the second source operand and the first source operand and returns the result of the comparison to the destination operand. The comparison predicate operand (immediate operand) specifies the type of comparison performed.

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 32-bit memory location. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged. The comparison result is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 32-bit memory location. The result is stored in the low 32 bits of the destination operand; bits 127:32 of the destination operand are copied from the first source operand. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. The comparison result is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

EVEX encoded version: The first source operand (second operand) is an XMM register. The second source operand can be a XMM register or a 32-bit memory location. The destination operand (first operand) is an opmask register. The comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false), written to the destination starting from the LSB according to the writemask k2. Bits (MAX_KL-1:128) of the destination register are cleared.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-8). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-8). Bits 3 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.01H:ECX.AVX[28] = 0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSS instruction, for processors with "CPUID.01H:ECX.AVX[28] = 0". See Table 3-15. The compiler should treat reserved imm8 values as illegal syntax.

Table 3-15. Pseudo-Op and CMPSS Implementation

Pseudo-Op	CMPSS Implementation
CMPEQSS xmm1, xmm2	CMPSS xmm1, xmm2, 0
CMPLTSS xmm1, xmm2	CMPSS xmm1, xmm2, 1
CMPLESS xmm1, xmm2	CMPSS xmm1, xmm2, 2
CMPUNORDSS xmm1, xmm2	CMPSS xmm1, xmm2, 3
CMPNEQSS xmm1, xmm2	CMPSS xmm1, xmm2, 4
CMPNLTSS xmm1, xmm2	CMPSS xmm1, xmm2, 5
CMPNLESS xmm1, xmm2	CMPSS xmm1, xmm2, 6
CMPORDSS xmm1, xmm2	CMPSS xmm1, xmm2, 7

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with "CPUID.01H:ECX.AVX[28] = 1" implement the full complement of 32 predicates shown in Table 3-14, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPSS instruction. See Table 3-16, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. The compiler should treat reserved imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPSS instructions in a similar fashion by extending the syntax listed in Table 3-16.

Table 3-16. Pseudo-Op and VCMPSS Implementation

Pseudo-Op	CMPSS Implementation
VCMPEQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 0
VCMPLTSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1
VCMPLESS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 2
VCMPUNORDSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 3
VCMPNEQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 4
VCMPNLTSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 5
VCMPNLESS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 6
VCMPORDSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 7
VCMPEQ_UQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 8
VCMPNGESS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 9
VCMPNGTSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 0AH
VCMPFALSESS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 0BH
VCMPNEQ_OQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 0CH

Table 3-16. Pseudo-Op and VCMPSS Implementation (Contd.)

Pseudo-Op	CMPSS Implementation
VCMPGESS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 0DH
VCMPGTSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 0EH
VCMPTRUESS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 0FH
VCMPEQ_OSSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 10H
VCMPLT_OQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 11H
VCMPLE_OQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 12H
VCMPUNORD_SSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 13H
VCMPNEQ_USSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 14H
VCMPNLT_UQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 15H
VCMPNLE_UQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 16H
VCMPORD_SSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 17H
VCMPEQ_USSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 18H
VCMPNGE_UQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 19H
VCMPNGT_UQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1AH
VCMPFALSE_OSSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1BH
VCMPNEQ_OSSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1CH
VCMPGE_OQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1DH
VCMPGT_OQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1EH
VCMPTRUE_USSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1FH

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

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP3 := EQ_OQ; OP5 := EQ_OQ;
    1: OP3 := LT_OS; OP5 := LT_OS;
    2: OP3 := LE_OS; OP5 := LE_OS;
    3: OP3 := UNORD_Q; OP5 := UNORD_Q;
    4: OP3 := NEQ_UQ; OP5 := NEQ_UQ;
    5: OP3 := NLT_US; OP5 := NLT_US;
    6: OP3 := NLE_US; OP5 := NLE_US;
    7: OP3 := ORD_Q; OP5 := ORD_Q;
    8: OP5 := EQ_UQ;
    9: OP5 := NGE_US;
    10: OP5 := NGT_US;
    11: OP5 := FALSE_OQ;
    12: OP5 := NEQ_OQ;
    13: OP5 := GE_OS;
    14: OP5 := GT_OS;
    15: OP5 := TRUE_UQ;
    16: OP5 := EQ_OS;
    17: OP5 := LT_OQ;
    18: OP5 := LE_OQ;
    19: OP5 := UNORD_S;
```

```
    20: OP5 := NEQ_US;
    21: OP5 := NLT_UQ;
    22: OP5 := NLE_UQ;
    23: OP5 := ORD_S;
    24: OP5 := EQ_US;
    25: OP5 := NGE_UQ;
    26: OP5 := NGT_UQ;
    27: OP5 := FALSE_OS;
    28: OP5 := NEQ_OS;
    29: OP5 := GE_OQ;
    30: OP5 := GT_OQ;
    31: OP5 := TRUE_US;
    DEFAULT: Reserved
ESAC;
```

VCMPSS (EVEX Encoded Version)

```
CMP0 := SRC1[31:0] OP5 SRC2[31:0];

IF k2[0] or *no writemask*
    THEN    IF CMP0 = TRUE
                THEN DEST[0] := 1;
                ELSE DEST[0] := 0; FI;
    ELSE    DEST[0] := 0                    ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] := 0
```

CMPSS (128-bit Legacy SSE Version)

```
CMP0 := DEST[31:0] OP3 SRC[31:0];
IF CMP0 = TRUE
THEN DEST[31:0] := FFFFFFFFH;
ELSE DEST[31:0] := 00000000H; FI;
DEST[MAXVL-1:32] (Unmodified)
```

VCMPSS (VEX.128 Encoded Version)

```
CMP0 := SRC1[31:0] OP5 SRC2[31:0];
IF CMP0 = TRUE
THEN DEST[31:0] := FFFFFFFFH;
ELSE DEST[31:0] := 00000000H; FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCMPSS __mmask8 _mm_cmp_ss_mask(__m128 a, __m128 b, int imm);
VCMPSS __mmask8 _mm_cmp_round_ss_mask(__m128 a, __m128 b, int imm, int sae);
VCMPSS __mmask8 _mm_mask_cmp_ss_mask(__mmask8 k1, __m128 a, __m128 b, int imm);
VCMPSS __mmask8 _mm_mask_cmp_round_ss_mask(__mmask8 k1, __m128 a, __m128 b, int imm, int sae);
(V)CMPSS __m128 _mm_cmp_ss(__m128 a, __m128 b, const int imm)

SIMD Floating-Point Exceptions

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in Table 3-8, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-49, "Type E3 Class Exception Conditions."

CMPXCHG—Compare and Exchange

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F B0/r	CMPXCHG r/m8[1], r8[1]	MR	Valid	Valid[2]	Compare AL with r/m8. If equal, ZF is set and r8 is loaded into r/m8. Else, clear ZF and load r/m8 into AL.
0F B1/r	CMPXCHG r/m16, r16	MR	Valid	Valid[2]	Compare AX with r/m16. If equal, ZF is set and r16 is loaded into r/m16. Else, clear ZF and load r/m16 into AX.
0F B1/r	CMPXCHG r/m32, r32	MR	Valid	Valid[2]	Compare EAX with r/m32. If equal, ZF is set and r32 is loaded into r/m32. Else, clear ZF and load r/m32 into EAX.
REX.W + 0F B1/r	CMPXCHG r/m64, r64	MR	Valid	N.E.	Compare RAX with r/m64. If equal, ZF is set and r64 is loaded into r/m64. Else, clear ZF and load r/m64 into RAX.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.
2. See the IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

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

Description

Compares the value in the AL, AX, EAX, or RAX register with the first operand (destination operand). If the two values are equal, the second operand (source operand) is loaded into the destination operand. Otherwise, the destination operand is loaded into the AL, AX, EAX or RAX register. RAX register is available only in 64-bit mode.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. To simplify the interface to the processor's bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

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

IA-32 Architecture Compatibility

This instruction is not supported on Intel processors earlier than the Intel486 processors.

Operation

```
(* Accumulator = AL, AX, EAX, or RAX depending on whether a byte, word, doubleword, or quadword comparison is being performed *)
TEMP := DEST
IF accumulator = TEMP
    THEN
        ZF := 1;
        DEST := SRC;
    ELSE
        ZF := 0;
        accumulator := TEMP;
        DEST := TEMP;
FI;
```

Flags Affected

The ZF flag is set if the values in the destination operand and register AL, AX, or EAX are equal; otherwise it is cleared. The CF, PF, AF, SF, and OF flags are set according to the results of the comparison operation.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F C7 /1 CMPXCHG8B m64	M	Valid	Valid*	Compare EDX:EAX with m64. If equal, set ZF and load ECX:EBX into m64. Else, clear ZF and load m64 into EDX:EAX.
REX.W + 0F C7 /1 CMPXCHG16B m128	M	Valid	N.E.	Compare RDX:RAX with m128. If equal, set ZF and load RCX:RBX into m128. Else, clear ZF and load m128 into RDX:RAX.

NOTES:
*See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	N/A	N/A	N/A

Description

Compares the 64-bit value in EDX:EAX (or 128-bit value in RDX:RAX if operand size is 128 bits) with the operand (destination operand). If the values are equal, the 64-bit value in ECX:EBX (or 128-bit value in RCX:RBX) is stored in the destination operand. Otherwise, the value in the destination operand is loaded into EDX:EAX (or RDX:RAX). The destination operand is an 8-byte memory location (or 16-byte memory location if operand size is 128 bits). For the EDX:EAX and ECX:EBX register pairs, EDX and ECX contain the high-order 32 bits and EAX and EBX contain the low-order 32 bits of a 64-bit value. For the RDX:RAX and RCX:RBX register pairs, RDX and RCX contain the high-order 64 bits and RAX and RBX contain the low-order 64bits of a 128-bit value.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. To simplify the interface to the processor's bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

In 64-bit mode, default operation size is 64 bits. Use of the REX.W prefix promotes operation to 128 bits. Note that CMPXCHG16B requires that the destination (memory) operand be 16-byte aligned. See the summary chart at the beginning of this section for encoding data and limits. For information on the CPUID flag that indicates CMPXCHG16B, see Chapter 21 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

IA-32 Architecture Compatibility

This instruction encoding is not supported on Intel processors earlier than the Pentium processors.

Operation

```
IF (64-Bit Mode and OperandSize = 64)
    THEN
        TEMP128 := DEST
        IF (RDX:RAX = TEMP128)
            THEN
                ZF := 1;
                DEST := RCX:RBX;
            ELSE
                ZF := 0;
                RDX:RAX := TEMP128;
                DEST := TEMP128;
                FI;
        FI
    ELSE
        TEMP64 := DEST;
        IF (EDX:EAX = TEMP64)
            THEN
                ZF := 1;
                DEST := ECX:EBX;
            ELSE
                ZF := 0;
                EDX:EAX := TEMP64;
                DEST := TEMP64;
                FI;
        FI;
FI;
```

Flags Affected

The ZF flag is set if the destination operand and EDX:EAX are equal; otherwise it is cleared. The CF, PF, AF, SF, and OF flags are unaffected.

Protected Mode Exceptions

#UD	If the destination is not a memory operand.
#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#UD	If the destination operand is not a memory location.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#UD	If the destination operand is not a memory location.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If memory operand for CMPXCHG16B is not aligned on a 16-byte boundary. If CPUID.01H:ECX.CMPXCHG16B[13] = 0.
#UD	If the destination operand is not a memory location.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2F /r COMISD xmm1, xmm2/m64	A	V/V	SSE2	Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
VEX.LIG.66.0F.WIG 2F /r VCOMISD xmm1, xmm2/m64	A	V/V	AVX	Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
EVEX.LLIG.66.0F.W1 2F /r VCOMISD xmm1, xmm2/m64{sae}	B	V/V	AVX512F OR AVX10.1	Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

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

Description

Compares the double precision floating-point values in the low quadwords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF, and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 64 bit memory location. The COMISD instruction differs from the UCOMISD instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The UCOMISD instruction signals an invalid operation exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

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

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

Operation

COMISD (All Versions)

```
RESULT :=    OrderedCompare(DEST[63:0] <> SRC[63:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
    UNORDERED: ZF,PF,CF := 111;
    GREATER_THAN: ZF,PF,CF := 000;
    LESS_THAN: ZF,PF,CF := 001;
    EQUAL: ZF,PF,CF := 100;
ESAC;
OF, AF, SF := 0; }
```

Intel C/C++ Compiler Intrinsic Equivalent

VCOMISD int _mm_comi_round_sd(__m128d a, __m128d b, int imm, int sae);
VCOMISD int _mm_comieq_sd (__m128d a, __m128d b)
VCOMISD int _mm_comilt_sd (__m128d a, __m128d b)
VCOMISD int _mm_comile_sd (__m128d a, __m128d b)
VCOMISD int _mm_comigt_sd (__m128d a, __m128d b)
VCOMISD int _mm_comige_sd (__m128d a, __m128d b)
VCOMISD int _mm_comineq_sd (__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-50, "Type E3NF Class Exception Conditions."
Additionally:

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

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 2F /r COMISS xmm1, xmm2/m32	A	V/V	SSE	Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
VEX.LIG.0F.WIG 2F /r VCOMISS xmm1, xmm2/m32	A	V/V	AVX	Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
EVEX.LLIG.0F.W0 2F /r VCOMISS xmm1, xmm2/m32{sae}	B	V/V	AVX512F OR AVX10.1	Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

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

Description

Compares the single precision floating-point values in the low quadwords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF, and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 32 bit memory location.

The COMISS instruction differs from the UCOMISS instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The UCOMISS instruction signals an invalid operation exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

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

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

Operation

COMISS (All Versions)

```
RESULT :=    OrderedCompare(DEST[31:0] <> SRC[31:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
    UNORDERED: ZF,PF,CF := 111;
    GREATER_THAN: ZF,PF,CF := 000;
    LESS_THAN: ZF,PF,CF := 001;
    EQUAL: ZF,PF,CF := 100;
ESAC;
OF, AF, SF := 0; }
```

Intel C/C++ Compiler Intrinsic Equivalent

VCOMISS int _mm_comi_round_ss(__m128 a, __m128 b, int imm, int sae);
VCOMISS int _mm_comieq_ss (__m128 a, __m128 b)
VCOMISS int _mm_comilt_ss (__m128 a, __m128 b)
VCOMISS int _mm_comile_ss (__m128 a, __m128 b)
VCOMISS int _mm_comigt_ss (__m128 a, __m128 b)
VCOMISS int _mm_comige_ss (__m128 a, __m128 b)
VCOMISS int _mm_comineq_ss (__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-50, "Type E3NF Class Exception Conditions."

Additionally:

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

CPUID—CPU Identification

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F A2	CPUID	ZO	Valid	Valid	Returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, as determined by input entered in EAX and, in some cases, ECX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

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 and, in some cases, ECX.

Chapter 21, "Processor Identification and Feature Determination," in Volume 1 of the Intel® 64 and IA-32 Architectures Software Developer's Manual provides CPUID leaf information and shows information returned, depending on the initial value loaded into the EAX and ECX registers.

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. Although the CPUID instruction provides serialization, it is not the preferred method on newer processors that support the SERIALIZE instruction. See "Serializing Instructions" in Chapter 11 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A for more details.

Execution of CPUID causes a VM exit when executed in VMX non-root operation. See Chapter 27, "Virtual Machine Control Structures," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C for more details.

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;
(* Note that for some leaf values in EAX, the subleaf value in ECX is ignored. *)
(* Note that for invalid CPUID leaves and subleaves, the output values returned in EAX, EBX, ECX, and EDX are "Reserved" *)
(* Refer to Volume 1, Chapter 21 for details surrounding CPUID_INFO() *)
(EAX, EBX, ECX, EDX) := CPUID_INFO(EAX, ECX)
```

Flags Affected

None.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

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

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.

CRC32—Accumulate CRC32 Value

Opcode/ Instruction		Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F2 0F 38 F0 /r	CRC32 r32, r/m8[1]	RM	Valid	Valid	Accumulate CRC32 on r/m8.
F2 0F 38 F1 /r	CRC32 r32, r/m16	RM	Valid	Valid	Accumulate CRC32 on r/m16.
F2 0F 38 F1 /r	CRC32 r32, r/m32	RM	Valid	Valid	Accumulate CRC32 on r/m32.
F2 REX.W 0F 38 F0 /r	CRC32 r64, r/m8	RM	Valid	N.E.	Accumulate CRC32 on r/m8.
F2 REX.W 0F 38 F1 /r	CRC32 r64, r/m64	RM	Valid	N.E.	Accumulate CRC32 on r/m64.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

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

Description

Starting with an initial value in the first operand (destination operand), accumulates a CRC32 (polynomial 11EDC6F41H) value for the second operand (source operand) and stores the result in the destination operand. The source operand can be a register or a memory location. The destination operand must be an r32 or r64 register. If the destination is an r64 register, then the 32-bit result is stored in the least significant double word and 00000000H is stored in the most significant double word of the r64 register.

The initial value supplied in the destination operand is a double word integer stored in the r32 register or the least significant double word of the r64 register. To incrementally accumulate a CRC32 value, software retains the result of the previous CRC32 operation in the destination operand, then executes the CRC32 instruction again with new input data in the source operand. Data contained in the source operand is processed in reflected bit order. This means that the most significant bit of the source operand is treated as the least significant bit of the quotient, and so on, for all the bits of the source operand. Likewise, the result of the CRC operation is stored in the destination operand in reflected bit order. This means that the most significant bit of the resulting CRC (bit 31) is stored in the least significant bit of the destination operand (bit 0), and so on, for all the bits of the CRC.

Operation

Notes:

```
BIT_REFLECT64: DST[63-0] = SRC[0-63]
BIT_REFLECT32: DST[31-0] = SRC[0-31]
BIT_REFLECT16: DST[15-0] = SRC[0-15]
BIT_REFLECT8: DST[7-0] = SRC[0-7]
MOD2: Remainder from Polynomial division modulus 2
```

CRC32 instruction for 64-bit source operand and 64-bit destination operand:

```
TEMP1[63-0] := BIT_REFLECT64 (SRC[63-0])
TEMP2[31-0] := BIT_REFLECT32 (DEST[31-0])
TEMP3[95-0] := TEMP1[63-0] « 32
TEMP4[95-0] := TEMP2[31-0] « 64
TEMP5[95-0] := TEMP3[95-0] XOR TEMP4[95-0]
TEMP6[31-0] := TEMP5[95-0] MOD2 11EDC6F41H
DEST[31-0] := BIT_REFLECT (TEMP6[31-0])
DEST[63-32] := 00000000H
```

CRC32 instruction for 32-bit source operand and 32-bit destination operand:

```
TEMP1[31-0] := BIT_REFLECT32 (SRC[31-0])
TEMP2[31-0] := BIT_REFLECT32 (DEST[31-0])
TEMP3[63-0] := TEMP1[31-0] « 32
TEMP4[63-0] := TEMP2[31-0] « 32
TEMP5[63-0] := TEMP3[63-0] XOR TEMP4[63-0]
TEMP6[31-0] := TEMP5[63-0] MOD2 11EDC6F41H
DEST[31-0] := BIT_REFLECT (TEMP6[31-0])
```

CRC32 instruction for 16-bit source operand and 32-bit destination operand:

```
TEMP1[15-0] := BIT_REFLECT16 (SRC[15-0])
TEMP2[31-0] := BIT_REFLECT32 (DEST[31-0])
TEMP3[47-0] := TEMP1[15-0] « 32
TEMP4[47-0] := TEMP2[31-0] « 16
TEMP5[47-0] := TEMP3[47-0] XOR TEMP4[47-0]
TEMP6[31-0] := TEMP5[47-0] MOD2 11EDC6F41H
DEST[31-0] := BIT_REFLECT (TEMP6[31-0])
```

CRC32 instruction for 8-bit source operand and 64-bit destination operand:

```
TEMP1[7-0] := BIT_REFLECT8(SRC[7-0])
TEMP2[31-0] := BIT_REFLECT32 (DEST[31-0])
TEMP3[39-0] := TEMP1[7-0] « 32
TEMP4[39-0] := TEMP2[31-0] « 8
TEMP5[39-0] := TEMP3[39-0] XOR TEMP4[39-0]
TEMP6[31-0] := TEMP5[39-0] MOD2 11EDC6F41H
DEST[31-0] := BIT_REFLECT (TEMP6[31-0])
DEST[63-32] := 00000000H
```

CRC32 instruction for 8-bit source operand and 32-bit destination operand:

```
TEMP1[7-0] := BIT_REFLECT8(SRC[7-0])
TEMP2[31-0] := BIT_REFLECT32 (DEST[31-0])
TEMP3[39-0] := TEMP1[7-0] « 32
TEMP4[39-0] := TEMP2[31-0] « 8
TEMP5[39-0] := TEMP3[39-0] XOR TEMP4[39-0]
TEMP6[31-0] := TEMP5[39-0] MOD2 11EDC6F41H
DEST[31-0] := BIT_REFLECT (TEMP6[31-0])
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

unsigned int _mm_crc32_u8(unsigned int crc, unsigned char data)
unsigned int _mm_crc32_u16(unsigned int crc, unsigned short data)
unsigned int _mm_crc32_u32(unsigned int crc, unsigned int data)
unsigned __int64 _mm_crc32_u64(unsigned __int64 crc, unsigned __int64 data)

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CPUID.01H:ECX.SSE4_2[20] = 0.
	If LOCK prefix is used.

Virtual 8086 Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If CPUID.01H:ECX.SSE4_2[20] = 0.
	If LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If CPUID.01H:ECX.SSE4_2[20] = 0.
	If LOCK prefix is used.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F E6 /r CVTDQ2PD xmm1, xmm2/m64	A	V/V	SSE2	Convert two packed signed doubleword integers from xmm2/mem to two packed double precision floating-point values in xmm1.
VEX.128.F3.0F.WIG E6 /r VCVTDQ2PD xmm1, xmm2/m64	A	V/V	AVX	Convert two packed signed doubleword integers from xmm2/mem to two packed double precision floating-point values in xmm1.
VEX.256.F3.0F.WIG E6 /r VCVTDQ2PD ymm1, xmm2/m128	A	V/V	AVX	Convert four packed signed doubleword integers from xmm2/mem to four packed double precision floating-point values in ymm1.
EVEX.128.F3.0F.W0 E6 /r VCVTDQ2PD xmm1 {k1}{z}, xmm2/m64/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert 2 packed signed doubleword integers from xmm2/m64/m32bcst to eight packed double precision floating-point values in xmm1 with writemask k1.
EVEX.256.F3.0F.W0 E6 /r VCVTDQ2PD ymm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert 4 packed signed doubleword integers from xmm2/m128/m32bcst to 4 packed double precision floating-point values in ymm1 with writemask k1.
EVEX.512.F3.0F.W0 E6 /r VCVTDQ2PD zmm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	AVX512F OR AVX10.1	Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed double precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

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

Description

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

EVEX encoded versions: The source operand can be a YMM/XMM/XMM (low 64 bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. Attempt to encode this instruction with EVEX embedded rounding is ignored.

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

VEX.128 encoded version: The source operand is an XMM register or 64- bit memory location. The destination operand is a XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

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

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

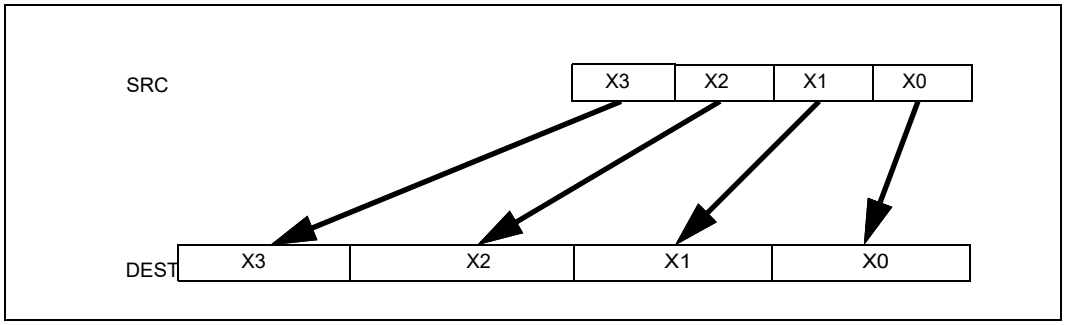


Figure 3-6. CVTDQ2PD (VEX.256 encoded version)

Operation

VCVTDQ2PD (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_Integer_To_Double_Precision_Floating_Point(SRC[k+31:k])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTDQ2PD (EVEX Encoded Versions) When SRC Operand is a Memory Source

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

FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
                ELSE
                    DEST[i+63:i] :=
            Convert_Integer_To_Double_Precision_Floating_Point(SRC[k+31:k])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
```

```
                    DEST[i+63:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTDQ2PD (VEX.256 Encoded Version)

```
DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[191:128] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[95:64])
DEST[255:192] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[127:96)
DEST[MAXVL-1:256] := 0
```

VCVTDQ2PD (VEX.128 Encoded Version)

```
DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] := 0
```

CVTDQ2PD (128-bit Legacy SSE Version)

```
DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTDQ2PD __m512d _mm512_cvtepi32_pd( __m256i a);
VCVTDQ2PD __m512d _mm512_mask_cvtepi32_pd( __m512d s, __mmask8 k, __m256i a);
VCVTDQ2PD __m512d _mm512_maskz_cvtepi32_pd( __mmask8 k, __m256i a);
VCVTDQ2PD __m256d _mm256_cvtepi32_pd (__m128i src);
VCVTDQ2PD __m256d _mm256_mask_cvtepi32_pd( __m256d s, __mmask8 k, __m256i a);
VCVTDQ2PD __m256d _mm256_maskz_cvtepi32_pd( __mmask8 k, __m256i a);
VCVTDQ2PD __m128d _mm_mask_cvtepi32_pd( __m128d s, __mmask8 k, __m128i a);
VCVTDQ2PD __m128d _mm_maskz_cvtepi32_pd( __mmask8 k, __m128i a);
CVTDQ2PD __m128d _mm_cvtepi32_pd (__m128i src)
```

Other Exceptions

VEX-encoded instructions, see Table 2-22, "Type 5 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-53, "Type E5 Class Exception Conditions."

Additionally:

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

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

Opcode Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5B /r CVTDQ2PS xmm1, xmm2/m128	A	V/V	SSE2	Convert four packed signed doubleword integers from xmm2/mem to four packed single precision floating-point values in xmm1.
VEX.128.0F.WIG 5B /r VCVTDQ2PS xmm1, xmm2/m128	A	V/V	AVX	Convert four packed signed doubleword integers from xmm2/mem to four packed single precision floating-point values in xmm1.
VEX.256.0F.WIG 5B /r VCVTDQ2PS ymm1, ymm2/m256	A	V/V	AVX	Convert eight packed signed doubleword integers from ymm2/mem to eight packed single precision floating-point values in ymm1.
EVEX.128.0F.W0 5B /r VCVTDQ2PS xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed signed doubleword integers from xmm2/m128/m32bcst to four packed single precision floating-point values in xmm1with writemask k1.
EVEX.256.0F.W0 5B /r VCVTDQ2PS ymm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed single precision floating-point values in ymm1with writemask k1.
EVEX.512.0F.W0 5B /r VCVTDQ2PS zmm1 {k1}{z}, zmm2/m512/m32bcst {er}	B	V/V	AVX512F OR AVX10.1	Convert sixteen packed signed doubleword integers from zmm2/m512/m32bcst to sixteen packed single precision floating-point values in zmm1with writemask k1.

Instruction Operand Encoding

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

Description

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

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

VEX.256 encoded version: The source operand is a YMM register or 256- bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding register destination are zeroed.

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

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

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

Operation

VCVTDQ2PS (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);  ; refer to Table 15-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);  ; refer to Table 15-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            Convert_Integer_To_Single_Precision_Floating_Point(SRC[i+31:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTDQ2PS (EVEX Encoded Versions) When SRC Operand is a Memory Source

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

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_Integer_To_Single_Precision_Floating_Point(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTDQ2PS (VEX.256 Encoded Version)
DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[127:96)
DEST[159:128] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[159:128])
DEST[191:160] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[191:160])
DEST[223:192] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[223:192])
DEST[255:224] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[255:224)
DEST[MAXVL-1:256] := 0

VCVTDQ2PS (VEX.128 Encoded Version)
DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[127z:96)
DEST[MAXVL-1:128] := 0

CVTDQ2PS (128-bit Legacy SSE Version)
DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[127z:96)
DEST[MAXVL-1:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VCVTDQ2PS __m512 _mm512_cvtepi32_ps(__m512i a);
VCVTDQ2PS __m512 _mm512_mask_cvtepi32_ps(__m512 s, __mmask16 k, __m512i a);
VCVTDQ2PS __m512 _mm512_maskz_cvtepi32_ps(__mmask16 k, __m512i a);
VCVTDQ2PS __m512 _mm512_cvt_roundepi32_ps(__m512i a, int r);
VCVTDQ2PS __m512 _mm512_mask_cvt_roundepi_ps(__m512 s, __mmask16 k, __m512i a, int r);
VCVTDQ2PS __m512 _mm512_maskz_cvt_roundepi32_ps(__mmask16 k, __m512i a, int r);
VCVTDQ2PS __m256 _mm256_mask_cvtepi32_ps(__m256 s, __mmask8 k, __m256i a);
VCVTDQ2PS __m256 _mm256_maskz_cvtepi32_ps(__mmask8 k, __m256i a);
VCVTDQ2PS __m128 _mm_mask_cvtepi32_ps(__m128 s, __mmask8 k, __m128i a);
VCVTDQ2PS __m128 _mm_maskz_cvtepi32_ps(__mmask8 k, __m128i a);
CVTDQ2PS __m256 _mm256_cvtepi32_ps (__m256i src)
CVTDQ2PS __m128 _mm_cvtepi32_ps (__m128i src)

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-19, "Type 2 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

Additionally:

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

CVTPD2DQ—Convert Packed Double Precision Floating-Point Values to Packed Doubleword Integers

Opcode Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F E6 /r CVTPD2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert two packed double precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1.
VEX.128.F2.0F.WIG E6 /r VCVTPD2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert two packed double precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1.
VEX.256.F2.0F.WIG E6 /r VCVTPD2DQ xmm1, ymm2/m256	A	V/V	AVX	Convert four packed double precision floating-point values in ymm2/mem to four signed doubleword integers in xmm1.
EVEX.128.F2.0F.W1 E6 /r VCVTPD2DQ xmm1 {k1}{z}, xmm2/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert two packed double precision floating-point values in xmm2/m128/m64bcst to two signed doubleword integers in xmm1 subject to writemask k1.
EVEX.256.F2.0F.W1 E6 /r VCVTPD2DQ xmm1 {k1}{z}, ymm2/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four signed doubleword integers in xmm1 subject to writemask k1.
EVEX.512.F2.0F.W1 E6 /r VCVTPD2DQ ymm1 {k1}{z}, zmm2/m512/m64bcst {er}	B	V/V	AVX512F OR AVX10.1	Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight signed doubleword integers in ymm1 subject to writemask k1.

Instruction Operand Encoding

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

Description

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

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. The upper bits (MAXVL-1:256/128/64) of the corresponding destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:64) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

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

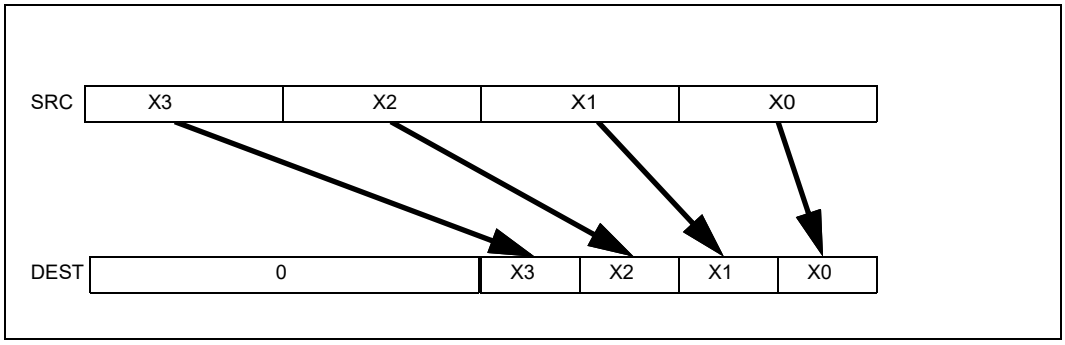


Figure 3-7. VCVTPD2DQ (VEX.256 encoded version)

Operation

VCVTPD2DQ (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_Integer(SRC[k+63:k])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

VCVTPD2DQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_Integer(SRC[k+63:k])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

VCVTPD2DQ (VEX.256 Encoded Version)

```
DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[95:64] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[191:128])
DEST[127:96] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[255:192)
DEST[MAXVL-1:128] := 0
```

VCVTPD2DQ (VEX.128 Encoded Version)

```
DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[MAXVL-1:64] := 0
```

CVTPD2DQ (128-bit Legacy SSE Version)

```
DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[127:64] := 0
DEST[MAXVL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPD2DQ __m256i _mm512_cvtpd_epi32(__m512d a);
VCVTPD2DQ __m256i _mm512_mask_cvtpd_epi32(__m256i s, __mmask8 k, __m512d a);
VCVTPD2DQ __m256i _mm512_maskz_cvtpd_epi32(__mmask8 k, __m512d a);
VCVTPD2DQ __m256i _mm512_cvt_roundpd_epi32(__m512d a, int r);
VCVTPD2DQ __m256i _mm512_mask_cvt_roundpd_epi32(__m256i s, __mmask8 k, __m512d a, int r);
VCVTPD2DQ __m256i _mm512_maskz_cvt_roundpd_epi32(__mmask8 k, __m512d a, int r);
VCVTPD2DQ __m128i _mm256_mask_cvtpd_epi32(__m128i s, __mmask8 k, __m256d a);
VCVTPD2DQ __m128i _mm256_maskz_cvtpd_epi32(__mmask8 k, __m256d a);
VCVTPD2DQ __m128i _mm_mask_cvtpd_epi32(__m128i s, __mmask8 k, __m128d a);
VCVTPD2DQ __m128i _mm_maskz_cvtpd_epi32(__mmask8 k, __m128d a);
VCVTPD2DQ __m128i _mm256_cvtpd_epi32 (__m256d src)
CVTPD2DQ __m128i _mm_cvtpd_epi32 (__m128d src)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Table 2-19, "Type 2 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

Additionally:

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

CVTPD2PI—Convert Packed Double Precision Floating-Point Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2D /r CVTPD2PI mm, xmm/m128	RM	V/V	SSE2	Convert two packed double precision floating-point values from xmm/m128 to two packed signed doubleword integers in mm.

Instruction Operand Encoding

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

Description

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

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

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

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPD2PI instruction is executed.

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

Operation

```
DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer32(SRC[63:0]);
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer32(SRC[127:64]);
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTPD1PI __m64 _mm_cvtpd_pi32(__m128d a)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

CVTPD2PS—Convert Packed Double Precision Floating-Point Values to Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5A /r CVTPD2PS xmm1, xmm2/m128	A	V/V	SSE2	Convert two packed double precision floating-point values in xmm2/mem to two single precision floating-point values in xmm1.
VEX.128.66.0F.WIG 5A /r VCVTPD2PS xmm1, xmm2/m128	A	V/V	AVX	Convert two packed double precision floating-point values in xmm2/mem to two single precision floating-point values in xmm1.
VEX.256.66.0F.WIG 5A /r VCVTPD2PS xmm1, ymm2/m256	A	V/V	AVX	Convert four packed double precision floating-point values in ymm2/mem to four single precision floating-point values in xmm1.
EVEX.128.66.0F.W1 5A /r VCVTPD2PS xmm1 {k1}{z}, xmm2/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert two packed double precision floating-point values in xmm2/m128/m64bcst to two single precision floating-point values in xmm1with writemask k1.
EVEX.256.66.0F.W1 5A /r VCVTPD2PS xmm1 {k1}{z}, ymm2/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four single precision floating-point values in xmm1with writemask k1.
EVEX.512.66.0F.W1 5A /r VCVTPD2PS ymm1 {k1}{z}, zmm2/m512/m64bcst {er}	B	V/V	AVX512F OR AVX10.1	Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight single precision floating-point values in ymm1with writemask k1.

Instruction Operand Encoding

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

Description

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

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64-bits) register conditionally updated with writemask k1. The upper bits (MAXVL-1:256/128/64) of the corresponding destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:64) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

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

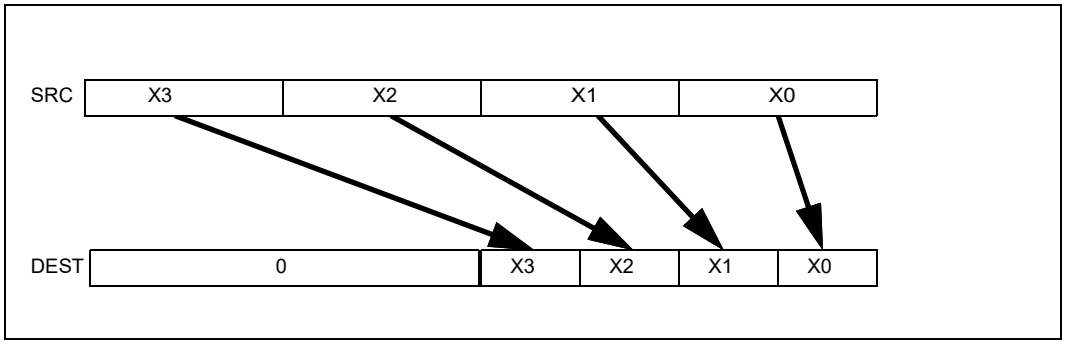


Figure 3-8. VCVTPD2PS (VEX.256 encoded version)

Operation

VCVTPD2PS (EVEX Encoded Version) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+31:i] := Convert_Double_Precision_Floating_Point_To_Single_Precision_Floating_Point(SRC[k+63:k])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

VCVTPD2PS (EVEX Encoded Version) When SRC Operand is a Memory Source
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=Convert_Double_Precision_Floating_Point_To_Single_Precision_Floating_Point(SRC[63:0])
                ELSE
                    DEST[i+31:i] := Convert_Double_Precision_Floating_Point_To_Single_Precision_Floating_Point(SRC[k+63:k])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

VCVTPD2PS (VEX.256 Encoded Version)

```
DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[95:64] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[191:128])
DEST[127:96] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[255:192)
DEST[MAXVL-1:128] := 0
```

VCVTPD2PS (VEX.128 Encoded Version)

```
DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[MAXVL-1:64] := 0
```

CVTPD2PS (128-bit Legacy SSE Version)

```
DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[127:64] := 0
DEST[MAXVL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPD2PS __m256 _mm512_cvtpd_ps(__m512d a);
VCVTPD2PS __m256 _mm512_mask_cvtpd_ps(__m256 s, __mmask8 k, __m512d a);
VCVTPD2PS __m256 _mm512_maskz_cvtpd_ps(__mmask8 k, __m512d a);
VCVTPD2PS __m256 _mm512_cvt_roundpd_ps(__m512d a, int r);
VCVTPD2PS __m256 _mm512_mask_cvt_roundpd_ps(__m256 s, __mmask8 k, __m512d a, int r);
VCVTPD2PS __m256 _mm512_maskz_cvt_roundpd_ps(__mmask8 k, __m512d a, int r);
VCVTPD2PS __m128 _mm256_mask_cvtpd_ps(__m128 s, __mmask8 k, __m256d a);
VCVTPD2PS __m128 _mm256_maskz_cvtpd_ps(__mmask8 k, __m256d a);
VCVTPD2PS __m128 _mm_mask_cvtpd_ps(__m128 s, __mmask8 k, __m128d a);
VCVTPD2PS __m128 _mm_maskz_cvtpd_ps(__mmask8 k, __m128d a);
VCVTPD2PS __m128 _mm256_cvtpd_ps (__m256d a)
CVTPD2PS __m128 _mm_cvtpd_ps (__m128d a)

SIMD Floating-Point Exceptions

Invalid, Precision, Underflow, Overflow, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, "Type 2 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

Additionally:

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

CVTPI2PD—Convert Packed Dword Integers to Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 2A /r CVTPI2PD xmm, mm/m64[1]	RM	Valid	Valid	Convert two packed signed doubleword integers from mm/mem64 to two packed double precision floating-point values in xmm.

NOTES:

1. Operation is different for different operand sets; see the Description section.

Instruction Operand Encoding

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

Description

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

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an XMM register. In addition, depending on the operand configuration:

- **For operands *xmm*, *mm*:** the instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPI2PD instruction is executed.
- **For operands *xmm*, *m64*:** the instruction does not cause a transition to MMX technology and does not take x87 FPU exceptions.

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

Operation

```
DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0]);
DEST[127:64] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32]);
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTPI2PD __m128d _mm_cvtpi32_pd(__m64 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

CVTPI2PS—Convert Packed Dword Integers to Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F 2A /r CVTPI2PS xmm, mm/m64	RM	Valid	Valid	Convert two signed doubleword integers from mm/m64 to two single precision floating-point values in xmm.

Instruction Operand Encoding

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

Description

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

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an XMM register. The results are stored in the low quadword of the destination operand, and the high quadword remains unchanged. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPI2PS instruction is executed.

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

Operation

```
DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
DEST[63:32] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32]);
(* High quadword of destination unchanged *)
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTPI2PS __m128 _mm_cvtpi32_ps(__m128 a, __m64 b)

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

See Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5B /r CVTPS2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert four packed single precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1.
VEX.128.66.0F.WIG 5B /r VCVTPS2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert four packed single precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1.
VEX.256.66.0F.WIG 5B /r VCVTPS2DQ ymm1, ymm2/m256	A	V/V	AVX	Convert eight packed single precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1.
EVEX.128.66.0F.W0 5B /r VCVTPS2DQ xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed doubleword values in xmm1 subject to writemask k1.
EVEX.256.66.0F.W0 5B /r VCVTPS2DQ ymm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed doubleword values in ymm1 subject to writemask k1.
EVEX.512.66.0F.W0 5B /r VCVTPS2DQ zmm1 {k1}{z}, zmm2/m512/m32bcst {er}	B	V/V	AVX512F OR AVX10.1	Convert sixteen packed single precision floating-point values from zmm2/m512/m32bcst to sixteen packed signed doubleword values in zmm1 subject to writemask k1.

Instruction Operand Encoding

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

Description

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

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

EVEX encoded versions: The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

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

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

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

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

Operation

VCVTPS2DQ (Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_Integer(SRC[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTPS2DQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

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

FOR j := 0 TO 15
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_Integer(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTPS2DQ (VEX.256 Encoded Version)

```
DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
DEST[63:32] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32])
DEST[95:64] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[95:64])
DEST[127:96] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[127:96)
DEST[159:128] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[159:128])
DEST[191:160] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[191:160])
DEST[223:192] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[223:192])
DEST[255:224] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[255:224])
```

VCVTPS2DQ (VEX.128 Encoded Version)

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

CVTPS2DQ (128-bit Legacy SSE Version)

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPS2DQ __m512i _mm512_cvtps_epi32(__m512 a);
VCVTPS2DQ __m512i _mm512_mask_cvtps_epi32(__m512i s, __mmask16 k, __m512 a);
VCVTPS2DQ __m512i _mm512_maskz_cvtps_epi32(__mmask16 k, __m512 a);
VCVTPS2DQ __m512i _mm512_cvt_roundps_epi32(__m512 a, int r);
VCVTPS2DQ __m512i _mm512_mask_cvt_roundps_epi32(__m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2DQ __m512i _mm512_maskz_cvt_roundps_epi32(__mmask16 k, __m512 a, int r);
VCVTPS2DQ __m256i _mm256_mask_cvtps_epi32(__m256i s, __mmask8 k, __m256 a);
VCVTPS2DQ __m256i _mm256_maskz_cvtps_epi32(__mmask8 k, __m256 a);
VCVTPS2DQ __m128i _mm_mask_cvtps_epi32(__m128i s, __mmask8 k, __m128 a);
VCVTPS2DQ __m128i _mm_maskz_cvtps_epi32(__mmask8 k, __m128 a);
VCVTPS2DQ __ m256i _mm256_cvtps_epi32 (__m256 a)
CVTPS2DQ __m128i _mm_cvtps_epi32 (__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-19, "Type 2 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

Additionally:

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

CVTPS2PD—Convert Packed Single Precision Floating-Point Values to Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5A /r CVTPS2PD xmm1, xmm2/m64	A	V/V	SSE2	Convert two packed single precision floating-point values in xmm2/m64 to two packed double precision floating-point values in xmm1.
VEX.128.0F.WIG 5A /r VCVTPS2PD xmm1, xmm2/m64	A	V/V	AVX	Convert two packed single precision floating-point values in xmm2/m64 to two packed double precision floating-point values in xmm1.
VEX.256.0F.WIG 5A /r VCVTPS2PD ymm1, xmm2/m128	A	V/V	AVX	Convert four packed single precision floating-point values in xmm2/m128 to four packed double precision floating-point values in ymm1.
EVEX.128.0F.W0 5A /r VCVTPS2PD xmm1 {k1}{z}, xmm2/m64/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert two packed single precision floating-point values in xmm2/m64/m32bcst to packed double precision floating-point values in xmm1 with writemask k1.
EVEX.256.0F.W0 5A /r VCVTPS2PD ymm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed single precision floating-point values in xmm2/m128/m32bcst to packed double precision floating-point values in ymm1 with writemask k1.
EVEX.512.0F.W0 5A /r VCVTPS2PD zmm1 {k1}{z}, ymm2/m256/m32bcst {sae}	B	V/V	AVX512F OR AVX10.1	Convert eight packed single precision floating-point values in ymm2/m256/b32bcst to eight packed double precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

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

Description

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

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64-bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 64- bit memory location. The destination operand is a XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

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

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

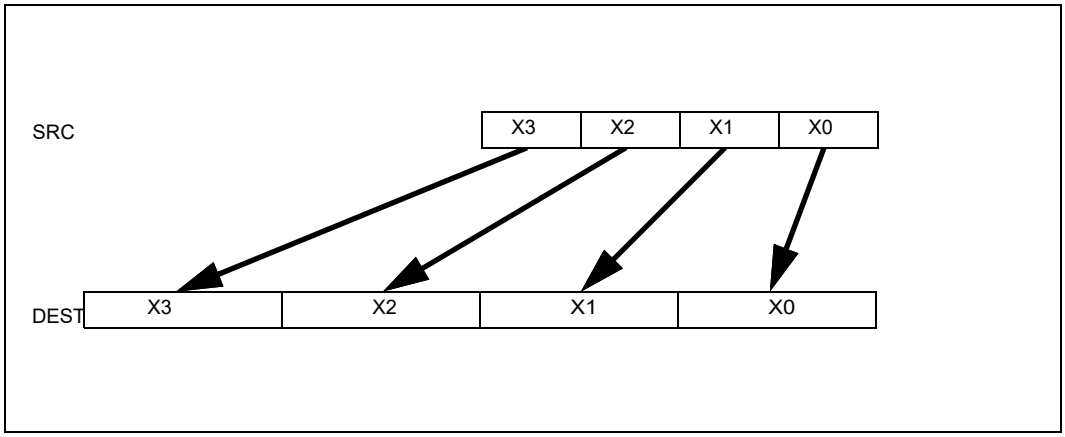


Figure 3-9. CVTPS2PD (VEX.256 encoded version)

Operation

VCVTPS2PD (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[k+31:k])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTPS2PD (EVEX Encoded Versions) When SRC Operand is a Memory Source

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

FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
                ELSE
                    DEST[i+63:i] :=
            Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[k+31:k])
            FI;
        ELSE
```

```
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTPS2PD (VEX.256 Encoded Version)

```
DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[191:128] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[95:64])
DEST[255:192] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[127:96)
DEST[MAXVL-1:256] := 0
```

VCVTPS2PD (VEX.128 Encoded Version)

```
DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] := 0
```

CVTPS2PD (128-bit Legacy SSE Version)

```
DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTPS2PD __m512d _mm512_cvtps_pd( __m256 a);
VCVTPS2PD __m512d _mm512_mask_cvtps_pd( __m512d s, __mmask8 k, __m256 a);
VCVTPS2PD __m512d _mm512_maskz_cvtps_pd( __mmask8 k, __m256 a);
VCVTPS2PD __m512d _mm512_cvt_roundps_pd( __m256 a, int sae);
VCVTPS2PD __m512d _mm512_mask_cvt_roundps_pd( __m512d s, __mmask8 k, __m256 a, int sae);
VCVTPS2PD __m512d _mm512_maskz_cvt_roundps_pd( __mmask8 k, __m256 a, int sae);
VCVTPS2PD __m256d _mm256_mask_cvtps_pd( __m256d s, __mmask8 k, __m128 a);
VCVTPS2PD __m256d _mm256_maskz_cvtps_pd( __mmask8 k, __m128a);
VCVTPS2PD __m128d _mm_mask_cvtps_pd( __m128d s, __mmask8 k, __m128 a);
VCVTPS2PD __m128d _mm_maskz_cvtps_pd( __mmask8 k, __m128 a);
VCVTPS2PD __m256d _mm256_cvtps_pd (__m128 a)
CVTPS2PD __m128d _mm_cvtps_pd (__m128 a)
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

Additionally:

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

CVTPS2PI—Convert Packed Single Precision Floating-Point Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F 2D /r CVTPS2PI mm, xmm/m64	RM	Valid	Valid	Convert two packed single precision floating-point values from xmm/m64 to two packed signed doubleword integers in mm.

Instruction Operand Encoding

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

Description

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

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register. When the source operand is an XMM register, the two single precision floating-point values are contained in the low quadword of the register. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

CVTPS2PI causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPS2PI instruction is executed.

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

Operation

```
DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
DEST[63:32] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32]);
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTPS2PI __m64 _mm_cvtps_pi32(__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

CVTSD2SI—Convert Scalar Double Precision Floating-Point Value to Signed Integer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 2D /r CVTSD2SI r32, xmm1/m64	A	V/V	SSE2	Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer r32.
F2 REX.W 0F 2D /r CVTSD2SI r64, xmm1/m64	A	V/N.E.	SSE2	Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.
VEX.LIG.F2.0F.W0 2D /r [1] VCVTSD2SI r32, xmm1/m64	A	V/V	AVX	Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer r32.
VEX.LIG.F2.0F.W1 2D /r [1] VCVTSD2SI r64, xmm1/m64	A	V/N.E.[2]	AVX	Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.
EVEX.LLIG.F2.0F.W0 2D /r VCVTSD2SI r32, xmm1/m64{er}	B	V/V	AVX512F OR AVX10.1	Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer r32.
EVEX.LLIG.F2.0F.W1 2D /r VCVTSD2SI r64, xmm1/m64{er}	B	V/N.E.[2]	AVX512F OR AVX10.1	Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.

NOTES:

1. Software should ensure VCVTSD2SI is encoded with VEX.L=0. Encoding VCVTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts a double precision floating-point value in the source operand (the second operand) to a signed integer in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000_00000000H is returned.

Legacy SSE instruction: Use of the REX.W prefix promotes the instruction to produce 64-bit data in 64-bit mode. See the summary chart at the beginning of this section for encoding data and limits.

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

Software should ensure VCVTSD2SI is encoded with VEX.L=0. Encoding VCVTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSD2SI (EVEX Encoded Version)

```
IF SRC *is register* AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-Bit Mode and OperandSize = 64
    THEN    DEST[63:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
    ELSE    DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
FI
```

(V)CVTSD2SI

```
IF 64-Bit Mode and OperandSize = 64
THEN
    DEST[63:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
ELSE
    DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSD2SI int _mm_cvtsd_i32(__m128d);
VCVTSD2SI int _mm_cvt_roundsd_i32(__m128d, int r);
VCVTSD2SI __int64 _mm_cvtsd_i64(__m128d);
VCVTSD2SI __int64 _mm_cvt_roundsd_i64(__m128d, int r);
CVTSD2SI __int64 _mm_cvtsd_si64(__m128d);
CVTSD2SI int _mm_cvtsd_si32(__m128d a)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-50, "Type E3NF Class Exception Conditions."

Additionally:

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

CVTSD2SS—Convert Scalar Double Precision Floating-Point Value to Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5A /r CVTSD2SS xmm1, xmm2/m64	A	V/V	SSE2	Convert one double precision floating-point value in xmm2/m64 to one single precision floating-point value in xmm1.
VEX.LIG.F2.0F.WIG 5A /r VCVTSD2SS xmm1,xmm2, xmm3/m64	B	V/V	AVX	Convert one double precision floating-point value in xmm3/m64 to one single precision floating-point value and merge with high bits in xmm2.
EVEX.LLIG.F2.0F.W1 5A /r VCVTSD2SS xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F OR AVX10.1	Convert one double precision floating-point value in xmm3/m64 to one single precision floating-point value and merge with high bits in xmm2 under writemask k1.

Instruction Operand Encoding

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

Description

Converts a double precision floating-point value in the "convert-from" source operand (the second operand in SSE2 version, otherwise the third operand) to a single precision floating-point value in the destination operand.

When the "convert-from" operand is an XMM register, the double precision floating-point value is contained in the low quadword of the register. The result is stored in the low doubleword of the destination operand. When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

128-bit Legacy SSE version: The "convert-from" source operand (the second operand) is an XMM register or memory location. Bits (MAXVL-1:32) of the corresponding destination register remain unchanged. The destination operand is an XMM register.

VEX.128 and EVEX encoded versions: The "convert-from" source operand (the third operand) can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: the converted result in written to the low doubleword element of the destination under the writemask.

Software should ensure VCVTSD2SS is encoded with VEX.L=0. Encoding VCVTSD2SS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSD2SS (EVEX Encoded Version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC2[63:0]);
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VCVTSD2SS (VEX.128 Encoded Version)

```
DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC2[63:0]);
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

CVTSD2SS (128-bit Legacy SSE Version)

```
DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0]);
(* DEST[MAXVL-1:32] Unmodified *)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSD2SS __m128 _mm_mask_cvtsd_ss(__m128 s, __mmask8 k, __m128 a, __m128d b);
VCVTSD2SS __m128 _mm_maskz_cvtsd_ss(__mmask8 k, __m128 a,__m128d b);
VCVTSD2SS __m128 _mm_cvt_roundsd_ss(__m128 a, __m128d b, int r);
VCVTSD2SS __m128 _mm_mask_cvt_roundsd_ss(__m128 s, __mmask8 k, __m128 a, __m128d b, int r);
VCVTSD2SS __m128 _mm_maskz_cvt_roundsd_ss(__mmask8 k, __m128 a,__m128d b, int r);
CVTSD2SS __m128_mm_cvtsd_ss(__m128 a, __m128d b)

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

CVTSI2SD—Convert Signed Integer to Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 2A /r CVTSI2SD xmm1, r32/m32	A	V/V	SSE2	Convert one signed doubleword integer from r32/m32 to one double precision floating-point value in xmm1.
F2 REX.W 0F 2A /r CVTSI2SD xmm1, r/m64	A	V/N.E.	SSE2	Convert one signed quadword integer from r/m64 to one double precision floating-point value in xmm1.
VEX.LIG.F2.0F.W0 2A /r VCVTSI2SD xmm1, xmm2, r/m32	B	V/V	AVX	Convert one signed doubleword integer from r/m32 to one double precision floating-point value in xmm1.
VEX.LIG.F2.0F.W1 2A /r VCVTSI2SD xmm1, xmm2, r/m64	B	V/N.E.[1]	AVX	Convert one signed quadword integer from r/m64 to one double precision floating-point value in xmm1.
EVEX.LLIG.F2.0F.W0 2A /r VCVTSI2SD xmm1, xmm2, r/m32	C	V/V	AVX512F OR AVX10.1	Convert one signed doubleword integer from r/m32 to one double precision floating-point value in xmm1.
EVEX.LLIG.F2.0F.W1 2A /r VCVTSI2SD xmm1, xmm2, r/m64{er}	C	V/N.E.[1]	AVX512F OR AVX10.1	Convert one signed quadword integer from r/m64 to one double precision floating-point value in xmm1.

NOTES:

1. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

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

Description

Converts a signed doubleword or quadword integer in the "convert-from" source operand to a double precision floating-point value in the destination operand. The result is stored in the low quadword of the destination operand, and the high quadword left unchanged. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: Use of the REX.W prefix promotes the instruction to 64-bit operands. The "convert-from" source operand (the second operand) is a general-purpose register or memory location. The destination is an XMM register Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: The "convert-from" source operand (the third operand) can be a general-purpose register or a memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.W0 version: attempt to encode this instruction with EVEX embedded rounding is ignored.

VEX.W1 and EVEX.W1 versions: promotes the instruction to use 64-bit input value in 64-bit mode.

Software should ensure VCVTSI2SD is encoded with VEX.L=0. Encoding VCVTSI2SD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSI2SD (EVEX Encoded Version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC2[63:0]);
ELSE
    DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC2[31:0]);
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VCVTSI2SD (VEX.128 Encoded Version)

```
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC2[63:0]);
ELSE
    DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC2[31:0]);
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

CVTSI2SD

```
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSI2SD __m128d _mm_cvti32_sd(__m128d s, int a);
VCVTSI2SD __m128d _mm_cvti64_sd(__m128d s, __int64 a);
VCVTSI2SD __m128d _mm_cvt_roundi64_sd(__m128d s, __int64 a, int r);
CVTSI2SD __m128d _mm_cvtsi64_sd(__m128d s, __int64 a);
CVTSI2SD __m128d_mm_cvtsi32_sd(__m128d a, int b)

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions," if W1; else see Table 2-22, "Type 5 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-50, "Type E3NF Class Exception Conditions," if W1; else see Table 2-61, "Type E10NF Class Exception Conditions."

CVTSI2SS—Convert Signed Integer to Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 2A /r CVTSI2SS xmm1, r/m32	A	V/V	SSE	Convert one signed doubleword integer from r/m32 to one single precision floating-point value in xmm1.
F3 REX.W 0F 2A /r CVTSI2SS xmm1, r/m64	A	V/N.E.	SSE	Convert one signed quadword integer from r/m64 to one single precision floating-point value in xmm1.
VEX.LIG.F3.0F.W0 2A /r VCVTSI2SS xmm1, xmm2, r/m32	B	V/V	AVX	Convert one signed doubleword integer from r/m32 to one single precision floating-point value in xmm1.
VEX.LIG.F3.0F.W1 2A /r VCVTSI2SS xmm1, xmm2, r/m64	B	V/N.E.[1]	AVX	Convert one signed quadword integer from r/m64 to one single precision floating-point value in xmm1.
EVEX.LLIG.F3.0F.W0 2A /r VCVTSI2SS xmm1, xmm2, r/m32{er}	C	V/V	AVX512F OR AVX10.1	Convert one signed doubleword integer from r/m32 to one single precision floating-point value in xmm1.
EVEX.LLIG.F3.0F.W1 2A /r VCVTSI2SS xmm1, xmm2, r/m64{er}	C	V/N.E.[1]	AVX512F OR AVX10.1	Convert one signed quadword integer from r/m64 to one single precision floating-point value in xmm1.

NOTES:

1. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

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

Description

Converts a signed doubleword or quadword integer in the "convert-from" source operand to a single precision floating-point value in the destination operand (first operand). The "convert-from" source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand, and the upper three doublewords are left unchanged. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

128-bit Legacy SSE version: In 64-bit mode, Use of the REX.W prefix promotes the instruction to use 64-bit input value. The "convert-from" source operand (the second operand) is a general-purpose register or memory location. Bits (MAXVL-1:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: The "convert-from" source operand (the third operand) can be a general-purpose register or a memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: the converted result in written to the low doubleword element of the destination under the writemask.

Software should ensure VCVTSI2SS is encoded with VEX.L=0. Encoding VCVTSI2SS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSI2SS (EVEX Encoded Version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VCVTSI2SS (VEX.128 Encoded Version)

```
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

CVTSI2SS (128-bit Legacy SSE Version)

```
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[31:0] :=Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSI2SS __m128 _mm_cvti32_ss(__m128 s, int a);
VCVTSI2SS __m128 _mm_cvt_roundi32_ss(__m128 s, int a, int r);
VCVTSI2SS __m128 _mm_cvti64_ss(__m128 s, __int64 a);
VCVTSI2SS __m128 _mm_cvt_roundi64_ss(__m128 s, __int64 a, int r);
CVTSI2SS __m128 _mm_cvtsi64_ss(__m128 s, __int64 a);
CVTSI2SS __m128 _mm_cvtsi32_ss(__m128 a, int b);

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-50, “Type E3NF Class Exception Conditions.”

CVTSS2SD—Convert Scalar Single Precision Floating-Point Value to Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5A /r CVTSS2SD xmm1, xmm2/m32	A	V/V	SSE2	Convert one single precision floating-point value in xmm2/m32 to one double precision floating-point value in xmm1.
VEX.LIG.F3.0F.WIG 5A /r VCVTSS2SD xmm1, xmm2, xmm3/m32	B	V/V	AVX	Convert one single precision floating-point value in xmm3/m32 to one double precision floating-point value and merge with high bits of xmm2.
EVEX.LLIG.F3.0F.W0 5A /r VCVTSS2SD xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	C	V/V	AVX512F OR AVX10.1	Convert one single precision floating-point value in xmm3/m32 to one double precision floating-point value and merge with high bits of xmm2 under writemask k1.

Instruction Operand Encoding

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

Description

Converts a single precision floating-point value in the "convert-from" source operand to a double precision floating-point value in the destination operand. When the "convert-from" source operand is an XMM register, the single precision floating-point value is contained in the low doubleword of the register. The result is stored in the low quadword of the destination operand.

128-bit Legacy SSE version: The "convert-from" source operand (the second operand) is an XMM register or memory location. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged. The destination operand is an XMM register.

VEX.128 and EVEX encoded versions: The "convert-from" source operand (the third operand) can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

Software should ensure VCVTSS2SD is encoded with VEX.L=0. Encoding VCVTSS2SD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSS2SD (EVEX Encoded Version)

```
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC2[31:0]);
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] = 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VCVTSS2SD (VEX.128 Encoded Version)

```
DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC2[31:0])
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

CVTSS2SD (128-bit Legacy SSE Version)

```
DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0]);
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSS2SD __m128d _mm_cvt_roundss_sd(__m128d a, __m128 b, int r);
VCVTSS2SD __m128d _mm_mask_cvt_roundss_sd(__m128d s, __mmask8 m, __m128d a,__m128 b, int r);
VCVTSS2SD __m128d _mm_maskz_cvt_roundss_sd(__mmask8 k, __m128d a, __m128 a, int r);
VCVTSS2SD __m128d _mm_mask_cvtss_sd(__m128d s, __mmask8 m, __m128d a,__m128 b);
VCVTSS2SD __m128d _mm_maskz_cvtss_sd(__mmask8 m, __m128d a,__m128 b);
CVTSS2SD __m128d_mm_cvtss_sd(__m128d a, __m128 a);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-49, "Type E3 Class Exception Conditions."

CVTSS2SI—Convert Scalar Single Precision Floating-Point Value to Signed Integer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 2D /r CVTSS2SI r32, xmm1/m32	A	V/V	SSE	Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.
F3 REX.W 0F 2D /r CVTSS2SI r64, xmm1/m32	A	V/N.E.	SSE	Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64.
VEX.LIG.F3.0F.W0 2D /r [1] VCVTSS2SI r32, xmm1/m32	A	V/V	AVX	Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.
VEX.LIG.F3.0F.W1 2D /r [1] VCVTSS2SI r64, xmm1/m32	A	V/N.E.[2]	AVX	Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64.
EVEX.LLIG.F3.0F.W0 2D /r VCVTSS2SI r32, xmm1/m32{er}	B	V/V	AVX512F OR AVX10.1	Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.
EVEX.LLIG.F3.0F.W1 2D /r VCVTSS2SI r64, xmm1/m32{er}	B	V/N.E.[2]	AVX512F OR AVX10.1	Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64.

NOTES:

1. Software should ensure VCVTSS2SI is encoded with VEX.L=0. Encoding VCVTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts a single precision floating-point value in the source operand (the second operand) to a signed integer in the destination operand (the first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000_00000000H is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to produce 64-bit data. See the summary chart at the beginning of this section for encoding data and limits.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

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

Software should ensure VCVTSS2SI is encoded with VEX.L=0. Encoding VCVTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSS2SI (EVEX Encoded Version)

```
IF (SRC *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-bit Mode and OperandSize = 64
THEN
    DEST[63:0] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
ELSE
    DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
FI;
```

(V)CVTSS2SI (Legacy and VEX.128 Encoded Version)

```
IF 64-bit Mode and OperandSize = 64
THEN
    DEST[63:0] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
ELSE
    DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSS2SI int _mm_cvtss_i32(__m128 a);
VCVTSS2SI int _mm_cvt_roundss_i32(__m128 a, int r);
VCVTSS2SI __int64 _mm_cvtss_i64(__m128 a);
VCVTSS2SI __int64 _mm_cvt_roundss_i64(__m128 a, int r);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions," additionally:

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Table 2-50, "Type E3NF Class Exception Conditions."

CVTTPD2DQ—Convert with Truncation Packed Double Precision Floating-Point Values to Packed Doubleword Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F E6 /r CVTTPD2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert two packed double precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1 using truncation.
VEX.128.66.0F.WIG E6 /r VCVTTPD2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert two packed double precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1 using truncation.
VEX.256.66.0F.WIG E6 /r VCVTTPD2DQ xmm1, ymm2/m256	A	V/V	AVX	Convert four packed double precision floating-point values in ymm2/mem to four signed doubleword integers in xmm1 using truncation.
EVEX.128.66.0F.W1 E6 /r VCVTTPD2DQ xmm1 {k1}{z}, xmm2/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert two packed double precision floating-point values in xmm2/m128/m64bcst to two signed doubleword integers in xmm1 using truncation subject to writemask k1.
EVEX.256.66.0F.W1 E6 /r VCVTTPD2DQ xmm1 {k1}{z}, ymm2/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four signed doubleword integers in xmm1 using truncation subject to writemask k1.
EVEX.512.66.0F.W1 E6 /r VCVTTPD2DQ ymm1 {k1}{z}, zmm2/m512/m64bcst {sae}	B	V/V	AVX512F OR AVX10.1	Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight signed doubleword integers in ymm1 using truncation subject to writemask k1.

Instruction Operand Encoding

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

Description

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

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1. The upper bits (MAXVL-1:256) of the corresponding destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:64) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

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

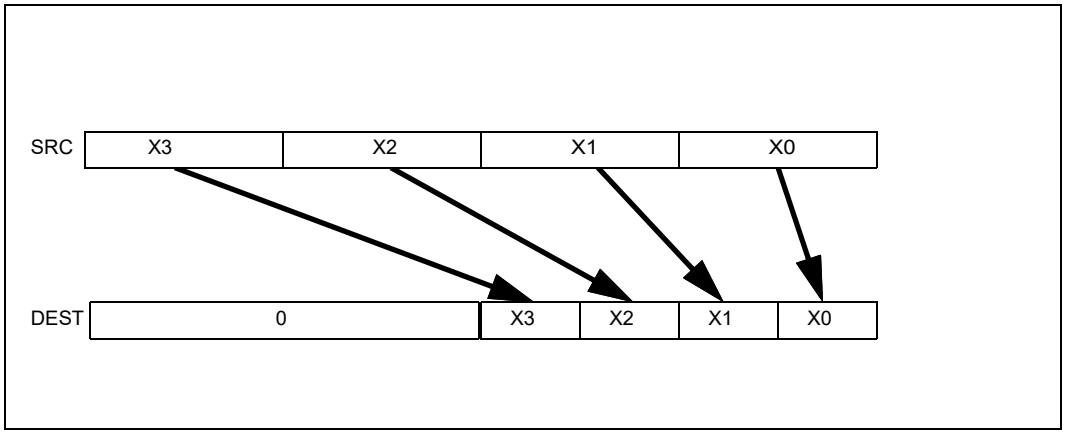


Figure 3-10. VCVTTPD2DQ (VEX.256 encoded version)

Operation

VCVTTPD2DQ (EVEX Encoded Versions) When SRC Operand is a Register

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

```
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[k+63:k])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

VCVTTPD2DQ (EVEX Encoded Versions) When SRC Operand is a Memory Source
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[k+63:k])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

VCVTTPD2DQ (VEX.256 Encoded Version)

```
DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[95:64] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[191:128])
DEST[127:96] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[255:192)
DEST[MAXVL-1:128] := 0
```

VCVTTPD2DQ (VEX.128 Encoded Version)

```
DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[MAXVL-1:64] := 0
```

CVTTPD2DQ (128-bit Legacy SSE Version)

```
DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[127:64] := 0
DEST[MAXVL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPD2DQ __m256i _mm512_cvttpd_epi32(__m512d a);
VCVTTPD2DQ __m256i _mm512_mask_cvttpd_epi32(__m256i s, __mmask8 k, __m512d a);
VCVTTPD2DQ __m256i _mm512_maskz_cvttpd_epi32(__mmask8 k, __m512d a);
VCVTTPD2DQ __m256i _mm512_cvtt_roundpd_epi32(__m512d a, int sae);
VCVTTPD2DQ __m256i _mm512_mask_cvtt_roundpd_epi32(__m256i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2DQ __m256i _mm512_maskz_cvtt_roundpd_epi32(__mmask8 k, __m512d a, int sae);
VCVTTPD2DQ __m128i _mm256_mask_cvttpd_epi32(__m128i s, __mmask8 k, __m256d a);
VCVTTPD2DQ __m128i _mm256_maskz_cvttpd_epi32(__mmask8 k, __m256d a);
VCVTTPD2DQ __m128i _mm_mask_cvttpd_epi32(__m128i s, __mmask8 k, __m128d a);
VCVTTPD2DQ __m128i _mm_maskz_cvttpd_epi32(__mmask8 k, __m128d a);
VCVTTPD2DQ __m128i _mm256_cvttpd_epi32 (__m256d src);
CVTTPD2DQ __m128i _mm_cvttpd_epi32 (__m128d src);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-19, "Type 2 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."
Additionally:

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

CVTTPD2PI—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 2C /r CVTTPD2PI mm, xmm/m128	RM	Valid	Valid	Convert two packer double precision floating-point values from xmm/m128 to two packed signed doubleword integers in mm using truncation.

Instruction Operand Encoding

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

Description

Converts two packed double precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand). The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTTPD2PI instruction is executed.

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

Operation

```
DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer32_Truncate(SRC[63:0]);
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer32_Truncate(SRC[127:64]);
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTTPD1PI __m64 _mm_cvttpd_pi32(__m128d a)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Mode Exceptions

See Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

CVTTPS2DQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Signed Doubleword Integer Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5B /r CVTTPS2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert four packed single precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1 using truncation.
VEX.128.F3.0F.WIG 5B /r VCVTTPS2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert four packed single precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1 using truncation.
VEX.256.F3.0F.WIG 5B /r VCVTTPS2DQ ymm1, ymm2/m256	A	V/V	AVX	Convert eight packed single precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1 using truncation.
EVEX.128.F3.0F.W0 5B /r VCVTTPS2DQ xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	AVX512VL AVX512F	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed doubleword values in xmm1 using truncation subject to writemask k1.
EVEX.256.F3.0F.W0 5B /r VCVTTPS2DQ ymm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	AVX512VL AVX512F	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed doubleword values in ymm1 using truncation subject to writemask k1.
EVEX.512.F3.0F.W0 5B /r VCVTTPS2DQ zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	B	V/V	AVX512F	Convert sixteen packed single precision floating-point values from zmm2/m512/m32bcst to sixteen packed signed doubleword values in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

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

Description

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

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

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

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

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

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

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

Operation

VCVTTPS2DQ (EVEX Encoded Versions) When SRC Operand is a Register

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

```
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTTPS2DQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

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

```
FOR j := 0 TO 15
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTTPS2DQ (VEX.256 Encoded Version)

```
DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
DEST[63:32] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32])
DEST[95:64] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[95:64])
DEST[127:96] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[127:96)
DEST[159:128] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[159:128])
DEST[191:160] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[191:160])
DEST[223:192] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[223:192])
DEST[255:224] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[255:224])
```

VCVTTPS2DQ (VEX.128 Encoded Version)

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

CVTTPS2DQ (128-bit Legacy SSE Version)

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPS2DQ __m512i _mm512_cvttps_epi32(__m512 a);
VCVTTPS2DQ __m512i _mm512_mask_cvttps_epi32(__m512i s, __mmask16 k, __m512 a);
VCVTTPS2DQ __m512i _mm512_maskz_cvttps_epi32(__mmask16 k, __m512 a);
VCVTTPS2DQ __m512i _mm512_cvtt_roundps_epi32(__m512 a, int sae);
VCVTTPS2DQ __m512i _mm512_mask_cvtt_roundps_epi32(__m512i s, __mmask16 k, __m512 a, int sae);
VCVTTPS2DQ __m512i _mm512_maskz_cvtt_roundps_epi32(__mmask16 k, __m512 a, int sae);
VCVTTPS2DQ __m256i _mm256_mask_cvttps_epi32(__m256i s, __mmask8 k, __m256 a);
VCVTTPS2DQ __m256i _mm256_maskz_cvttps_epi32(__mmask8 k, __m256 a);
VCVTTPS2DQ __m128i _mm_mask_cvttps_epi32(__m128i s, __mmask8 k, __m128 a);
VCVTTPS2DQ __m128i _mm_maskz_cvttps_epi32(__mmask8 k, __m128 a);
VCVTTPS2DQ __m256i _mm256_cvttps_epi32 (__m256 a)
CVTTPS2DQ __m128i _mm_cvttps_epi32 (__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-19, "Type 2 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."
Additionally:

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

CVTTPS2PI—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F 2C /r CVTTPS2PI mm, xmm/m64	RM	Valid	Valid	Convert two single precision floating-point values from xmm/m64 to two signed doubleword signed integers in mm using truncation.

Instruction Operand Encoding

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

Description

Converts two packed single precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is an MMX technology register. When the source operand is an XMM register, the two single precision floating-point values are contained in the low quadword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTTPS2PI instruction is executed.

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

Operation

```
DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0]);
DEST[63:32] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32]);
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTTPS2PI __m64 _mm_cvttps_pi32(__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

CVTTSD2SI—Convert With Truncation Scalar Double Precision Floating-Point Value to Signed Integer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 2C /r CVTTSD2SI r32, xmm1/m64	A	V/V	SSE2	Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.
F2 REX.W 0F 2C /r CVTTSD2SI r64, xmm1/m64	A	V/N.E.	SSE2	Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.
VEX.LIG.F2.0F.W0 2C /r [1] VCVTTSD2SI r32, xmm1/m64	A	V/V	AVX	Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.
VEX.LIG.F2.0F.W1 2C /r [1] VCVTTSD2SI r64, xmm1/m64	B	V/N.E.[2]	AVX	Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.
EVEX.LLIG.F2.0F.W0 2C /r VCVTTSD2SI r32, xmm1/m64{sae}	B	V/V	AVX512F OR AVX10.1	Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.
EVEX.LLIG.F2.0F.W1 2C /r VCVTTSD2SI r64, xmm1/m64{sae}	B	V/N.E.[2]	AVX512F OR AVX10.1	Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.

NOTES:

1. Software should ensure VCVTTSD2SI is encoded with VEX.L=0. Encoding VCVTTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts a double precision floating-point value in the source operand (the second operand) to a signed double-word integer (or signed quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general purpose register. When the source operand is an XMM register, the double precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000_00000000H is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

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

Software should ensure VCVTTSD2SI is encoded with VEX.L=0. Encoding VCVTTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

(V)CVTTSD2SI (All Versions)

```
IF 64-Bit Mode and OperandSize = 64
THEN
    DEST[63:0] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0]);
ELSE
    DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0]);
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTSD2SI int _mm_cvttsd_i32(__m128d a);
VCVTTSD2SI int _mm_cvtt_roundsd_i32(__m128d a, int sae);
VCVTTSD2SI __int64 _mm_cvttsd_i64(__m128d a);
VCVTTSD2SI __int64 _mm_cvtt_roundsd_i64(__m128d a, int sae);
CVTTSD2SI int _mm_cvttsd_si32(__m128d a);
CVTTSD2SI __int64 _mm_cvttsd_si64(__m128d a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions," additionally:

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Table 2-50, "Type E3NF Class Exception Conditions."

CVTTSS2SI—Convert With Truncation Scalar Single Precision Floating-Point Value to Signed Integer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 2C /r CVTTSS2SI r32, xmm1/m32	A	V/V	SSE	Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.
F3 REX.W 0F 2C /r CVTTSS2SI r64, xmm1/m32	A	V/N.E.	SSE	Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.
VEX.LIG.F3.0F.W0 2C /r [1] VCVTTSS2SI r32, xmm1/m32	A	V/V	AVX	Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.
VEX.LIG.F3.0F.W1 2C /r [1] VCVTTSS2SI r64, xmm1/m32	A	V/N.E.[2]	AVX	Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.
EVEX.LLIG.F3.0F.W0 2C /r VCVTTSS2SI r32, xmm1/m32{sae}	B	V/V	AVX512F OR AVX10.1	Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.
EVEX.LLIG.F3.0F.W1 2C /r VCVTTSS2SI r64, xmm1/m32{sae}	B	V/N.E.[2]	AVX512F OR AVX10.1	Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.

NOTES:

1. Software should ensure VCVTTSS2SI is encoded with VEX.L=0. Encoding VCVTTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts a single precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a 32-bit memory location. The destination operand is a general purpose register. When the source operand is an XMM register, the single precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000_00000000H is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

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

Software should ensure VCVTTSS2SI is encoded with VEX.L=0. Encoding VCVTTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

(V)CVTTSS2SI (All Versions)

```
IF 64-Bit Mode and OperandSize = 64
THEN
    DEST[63:0] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0]);
ELSE
    DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0]);
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTSS2SI int _mm_cvttss_i32(__m128 a);
VCVTTSS2SI int _mm_cvtt_roundss_i32(__m128 a, int sae);
VCVTTSS2SI __int64 _mm_cvttss_i64(__m128 a);
VCVTTSS2SI __int64 _mm_cvtt_roundss_i64(__m128 a, int sae);
CVTTSS2SI int _mm_cvttss_si32(__m128 a);
CVTTSS2SI __int64 _mm_cvttss_si64(__m128 a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Table 2-20, "Type 3 Class Exception Conditions," additionally:

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Table 2-50, "Type E3NF Class Exception Conditions."

CWD/CDQ/CQO—Convert Word to Doubleword/Convert Doubleword to Quadword

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
99	CWD	ZO	Valid	Valid	DX:AX := sign-extend of AX.
99	CDQ	ZO	Valid	Valid	EDX:EAX := sign-extend of EAX.
REX.W + 99	CQO	ZO	Valid	N.E.	RDX:RAX:= sign-extend of RAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Doubles the size of the operand in register AX, EAX, or RAX (depending on the operand size) by means of sign extension and stores the result in registers DX:AX, EDX:EAX, or RDX:RAX, respectively. The CWD instruction copies the sign (bit 15) of the value in the AX register into every bit position in the DX register. The CDQ instruction copies the sign (bit 31) of the value in the EAX register into every bit position in the EDX register. The CQO instruction (available in 64-bit mode only) copies the sign (bit 63) of the value in the RAX register into every bit position in the RDX register.

The CWD instruction can be used to produce a doubleword dividend from a word before word division. The CDQ instruction can be used to produce a quadword dividend from a doubleword before doubleword division. The CQO instruction can be used to produce a double quadword dividend from a quadword before a quadword division.

The CWD and CDQ mnemonics reference the same opcode. The CWD instruction is intended for use when the operand-size attribute is 16 and the CDQ instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when CWD is used and to 32 when CDQ is used. Others may treat these mnemonics as synonyms (CWD/CDQ) and use the current setting of the operand-size attribute to determine the size of values to be converted, regardless of the mnemonic used.

In 64-bit mode, use of the REX.W prefix promotes operation to 64 bits. The CQO mnemonics reference the same opcode as CWD/CDQ. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF OperandSize = 16 (* CWD instruction *)
    THEN
        DX := SignExtend(AX);
    ELSE IF OperandSize = 32 (* CDQ instruction *)
        EDX := SignExtend(EAX); FI;
    ELSE IF 64-Bit Mode and OperandSize = 64 (* CQO instruction*)
        RDX := SignExtend(RAX); FI;
FI;
```

Flags Affected

None.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

DAA—Decimal Adjust AL After Addition

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
27	DAA	ZO	Invalid	Valid	Decimal adjust AL after addition.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Adjusts the sum of two packed BCD values to create a packed BCD result. The AL register is the implied source and destination operand. The DAA instruction is only useful when it follows an ADD instruction that adds (binary addition) two 2-digit, packed BCD values and stores a byte result in the AL register. The DAA instruction then adjusts the contents of the AL register to contain the correct 2-digit, packed BCD result. If a decimal carry is detected, the CF and AF flags are set accordingly.

This instruction executes as described above in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

```
IF 64-Bit Mode
    THEN
        #UD;
    ELSE
        old_AL := AL;
        old_CF := CF;
        CF := 0;
        IF (((AL AND 0FH) > 9) or AF = 1)
            THEN
                AL := AL + 6;
                CF := old_CF or (Carry from AL := AL + 6);
                AF := 1;
            ELSE
                AF := 0;
        FI;
        IF ((old_AL > 99H) or (old_CF = 1))
            THEN
                AL := AL + 60H;
                CF := 1;
            ELSE
                CF := 0;
        FI;
FI;
```

Example

```
ADD     AL, BL    Before: AL=79H BL=35H EFLAGS(OSZAPC)=XXXXXX
                  After: AL=AEH BL=35H EFLAGS(OSZAPC)=110000
DAA               Before: AL=AEH BL=35H EFLAGS(OSZAPC)=110000
                  After: AL=14H BL=35H EFLAGS(OSZAPC)=X00111
DAA               Before: AL=2EH BL=35H EFLAGS(OSZAPC)=110000
                  After: AL=34H BL=35H EFLAGS(OSZAPC)=X00101
```

Flags Affected

The CF and AF flags are set if the adjustment of the value results in a decimal carry in either digit of the result (see the "Operation" section above). The SF, ZF, and PF flags are set according to the result. The OF flag is undefined.

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 in 64-bit mode.

DAS—Decimal Adjust AL After Subtraction

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
2F	DAS	ZO	Invalid	Valid	Decimal adjust AL after subtraction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Adjusts the result of the subtraction of two packed BCD values to create a packed BCD result. The AL register is the implied source and destination operand. The DAS instruction is only useful when it follows a SUB instruction that subtracts (binary subtraction) one 2-digit, packed BCD value from another and stores a byte result in the AL register. The DAS instruction then adjusts the contents of the AL register to contain the correct 2-digit, packed BCD result. If a decimal borrow is detected, the CF and AF flags are set accordingly.

This instruction executes as described above in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

```
IF 64-Bit Mode
    THEN
        #UD;
    ELSE
        old_AL := AL;
        old_CF := CF;
        CF := 0;
        IF (((AL AND 0FH) > 9) or AF = 1)
            THEN
                AL := AL - 6;
                CF := old_CF or (Borrow from AL := AL – 6);
                AF := 1;
            ELSE
                AF := 0;
        FI;
        IF ((old_AL > 99H) or (old_CF = 1))
            THEN
                AL := AL – 60H;
                CF := 1;
        FI;
FI;
```

Example

```
SUB     AL, BL    Before: AL = 35H, BL = 47H, EFLAGS(OSZAPC) = XXXXXX
                  After: AL = EEH, BL = 47H, EFLAGS(OSZAPC) = 010111
DAA               Before: AL = EEH, BL = 47H, EFLAGS(OSZAPC) = 010111
                  After: AL = 88H, BL = 47H, EFLAGS(OSZAPC) = X10111
```

Flags Affected

The CF and AF flags are set if the adjustment of the value results in a decimal borrow in either digit of the result (see the "Operation" section above). The SF, ZF, and PF flags are set according to the result. The OF flag is undefined.

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 in 64-bit mode.

DEC—Decrement by 1

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FE /1	DEC r/m8[1]	M	Valid	Valid	Decrement *r/m8* by 1.
FF /1	DEC r/m16	M	Valid	Valid	Decrement *r/m16* by 1.
FF /1	DEC r/m32	M	Valid	Valid	Decrement *r/m32* by 1.
REX.W + FF /1	DEC r/m64	M	Valid	N.E.	Decrement *r/m64* by 1.
48+rw	DEC r16	O	N.E.	Valid	Decrement *r16* by 1.
48+rd	DEC r32	O	N.E.	Valid	Decrement *r32* by 1.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	N/A	N/A	N/A
O	opcode + rd (r, w)	N/A	N/A	N/A

Description

Subtracts 1 from the destination operand, while preserving the state of the CF flag. The destination operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (To perform a decrement operation that updates the CF flag, use a SUB instruction with an immediate operand of 1.)

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, DEC r16 and DEC r32 are not encodable (because opcodes 48H through 4FH are REX prefixes). Otherwise, the instruction's 64-bit mode default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST := DEST – 1;
```

Flags Affected

The CF flag is not affected. The OF, SF, ZF, AF, and PF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination operand is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

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

#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

DIV—Unsigned Divide

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /6	DIV r/m8[1]	M	Valid	Valid	Unsigned divide AX by r/m8, with result stored in AL := Quotient, AH := Remainder.
F7 /6	DIV r/m16	M	Valid	Valid	Unsigned divide DX:AX by r/m16, with result stored in AX := Quotient, DX := Remainder.
F7 /6	DIV r/m32	M	Valid	Valid	Unsigned divide EDX:EAX by r/m32, with result stored in EAX := Quotient, EDX := Remainder.
REX.W + F7 /6	DIV r/m64	M	Valid	N.E.	Unsigned divide RDX:RAX by r/m64, with result stored in RAX := Quotient, RDX := Remainder.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

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

Description

Divides unsigned the value in the AX, DX:AX, EDX:EAX, or RDX:RAX registers (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, EDX:EAX, or RDX:RAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size (dividend/divisor). Division using 64-bit operand is available only in 64-bit mode.

Non-integral results are truncated (chopped) towards 0. The remainder is always less than the divisor in magnitude. Overflow is indicated with the #DE (divide error) exception rather than with the CF flag.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. In 64-bit mode when REX.W is applied, the instruction divides the unsigned value in RDX:RAX by the source operand and stores the quotient in RAX, the remainder in RDX.

See the summary chart at the beginning of this section for encoding data and limits. See Table 3-17.

Table 3-17. DIV Action

Operand Size	Dividend	Divisor	Quotient	Remainder	Maximum Quotient
Word/byte	AX	r/m8	AL	AH	255
Doubleword/word	DX:AX	r/m16	AX	DX	65,535
Quadword/doubleword	EDX:EAX	r/m32	EAX	EDX	$2^{32} - 1$
Doublequadword/ quadword	RDX:RAX	r/m64	RAX	RDX	$2^{64} - 1$

Operation

```
IF SRC = 0
    THEN #DE; FI; (* Divide Error *)
IF OperandSize = 8 (* Word/Byte Operation *)
    THEN
        temp := AX / SRC;
        IF temp > FFH
            THEN #DE; (* Divide error *)
            ELSE
                AL := temp;
                AH := AX MOD SRC;
        FI;
    ELSE IF OperandSize = 16 (* Doubleword/word operation *)
        THEN
            temp := DX:AX / SRC;
            IF temp > FFFFH
                THEN #DE; (* Divide error *)
            ELSE
                AX := temp;
                DX := DX:AX MOD SRC;
            FI;
        FI;
    ELSE IF Operandsize = 32 (* Quadword/doubleword operation *)
        THEN
            temp := EDX:EAX / SRC;
            IF temp > FFFFFFFFH
                THEN #DE; (* Divide error *)
            ELSE
                EAX := temp;
                EDX := EDX:EAX MOD SRC;
            FI;
        FI;
    ELSE IF 64-Bit Mode and Operandsize = 64 (* Doublequadword/quadword operation *)
        THEN
            temp := RDX:RAX / SRC;
            IF temp > FFFFFFFFFFFFFFFFH
                THEN #DE; (* Divide error *)
            ELSE
                RAX := temp;
                RDX := RDX:RAX MOD SRC;
            FI;
        FI;
FI;
```

Flags Affected

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

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#DE	If the source operand (divisor) is 0.
	If the quotient is too large for the designated register.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

DIVPD—Divide Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5E /r DIVPD xmm1, xmm2/m128	A	V/V	SSE2	Divide packed double precision floating-point values in xmm1 by packed double precision floating-point values in xmm2/mem.
VEX.128.66.0F.WIG 5E /r VDIVPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Divide packed double precision floating-point values in xmm2 by packed double precision floating-point values in xmm3/mem.
VEX.256.66.0F.WIG 5E /r VDIVPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Divide packed double precision floating-point values in ymm2 by packed double precision floating-point values in ymm3/mem.
EVEX.128.66.0F.W1 5E /r VDIVPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Divide packed double precision floating-point values in xmm2 by packed double precision floating-point values in xmm3/m128/m64bcst and write results to xmm1 subject to writemask k1.
EVEX.256.66.0F.W1 5E /r VDIVPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Divide packed double precision floating-point values in ymm2 by packed double precision floating-point values in ymm3/m256/m64bcst and write results to ymm1 subject to writemask k1.
EVEX.512.66.0F.W1 5E /r VDIVPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	C	V/V	AVX512F OR AVX10.1	Divide packed double precision floating-point values in zmm2 by packed double precision floating-point values in zmm3/m512/m64bcst and write results to zmm1 subject to writemask k1.

Instruction Operand Encoding

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

Description

Performs a SIMD divide of the double precision floating-point values in the first source operand by the floating-point values in the second source operand (the third operand). Results are written to the destination operand (the first operand).

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

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

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

128-bit Legacy SSE version: The second source operand (the second operand) can be an XMM register or an 128-bit memory location. The destination is the same as the first source operand. The upper bits (MAXVL-1:128) of the corresponding destination are unmodified.

Operation

VDIVPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);  ; refer to Table 15-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := SRC1[i+63:i] / SRC2[63:0]
                ELSE
                    DEST[i+63:i] := SRC1[i+63:i] / SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VDIVPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0] / SRC2[63:0]
DEST[127:64] := SRC1[127:64] / SRC2[127:64]
DEST[191:128] := SRC1[191:128] / SRC2[191:128]
DEST[255:192] := SRC1[255:192] / SRC2[255:192]
DEST[MAXVL-1:256] := 0;
```

VDIVPD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] / SRC2[63:0]
DEST[127:64] := SRC1[127:64] / SRC2[127:64]
DEST[MAXVL-1:128] := 0;
```

DIVPD (128-bit Legacy SSE Version)

```
DEST[63:0] := SRC1[63:0] / SRC2[63:0]
DEST[127:64] := SRC1[127:64] / SRC2[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VDIVPD __m512d _mm512_div_pd(__m512d a, __m512d b);
VDIVPD __m512d _mm512_mask_div_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VDIVPD __m512d _mm512_maskz_div_pd(__mmask8 k, __m512d a, __m512d b);
VDIVPD __m256d _mm256_mask_div_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VDIVPD __m256d _mm256_maskz_div_pd(__mmask8 k, __m256d a, __m256d b);
VDIVPD __m128d _mm_mask_div_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VDIVPD __m128d _mm_maskz_div_pd(__mmask8 k, __m128d a, __m128d b);
VDIVPD __m512d _mm512_div_round_pd(__m512d a, __m512d b, int);
VDIVPD __m512d _mm512_mask_div_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VDIVPD __m512d _mm512_maskz_div_round_pd(__mmask8 k, __m512d a, __m512d b, int);
VDIVPD __m256d _mm256_div_pd (__m256d a, __m256d b);
DIVPD __m128d _mm_div_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, "Type 2 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

DIVPS—Divide Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5E /r DIVPS xmm1, xmm2/m128	A	V/V	SSE	Divide packed single precision floating-point values in xmm1 by packed single precision floating-point values in xmm2/mem.
VEX.128.0F.WIG 5E /r VDIVPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Divide packed single precision floating-point values in xmm2 by packed single precision floating-point values in xmm3/mem.
VEX.256.0F.WIG 5E /r VDIVPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Divide packed single precision floating-point values in ymm2 by packed single precision floating-point values in ymm3/mem.
EVEX.128.0F.W0 5E /r VDIVPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Divide packed single precision floating-point values in xmm2 by packed single precision floating-point values in xmm3/m128/m32bcst and write results to xmm1 subject to writemask k1.
EVEX.256.0F.W0 5E /r VDIVPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Divide packed single precision floating-point values in ymm2 by packed single precision floating-point values in ymm3/m256/m32bcst and write results to ymm1 subject to writemask k1.
EVEX.512.0F.W0 5E /r VDIVPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	C	V/V	AVX512F OR AVX10.1	Divide packed single precision floating-point values in zmm2 by packed single precision floating-point values in zmm3/m512/m32bcst and write results to zmm1 subject to writemask k1.

Instruction Operand Encoding

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

Description

Performs a SIMD divide of the four, eight or sixteen packed single precision floating-point values in the first source operand (the second operand) by the four, eight or sixteen packed single precision floating-point values in the second source operand (the third operand). Results are written to the destination operand (the first operand).

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

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

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

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

Operation

VDIVPS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] := SRC1[i+31:i] / SRC2[31:0]
                ELSE
                    DEST[i+31:i] := SRC1[i+31:i] / SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VDIVPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[31:0] / SRC2[31:0]
DEST[63:32] := SRC1[63:32] / SRC2[63:32]
DEST[95:64] := SRC1[95:64] / SRC2[95:64]
DEST[127:96] := SRC1[127:96] / SRC2[127:96]
DEST[159:128] := SRC1[159:128] / SRC2[159:128]
DEST[191:160] := SRC1[191:160] / SRC2[191:160]
DEST[223:192] := SRC1[223:192] / SRC2[223:192]
DEST[255:224] := SRC1[255:224] / SRC2[255:224].
DEST[MAXVL-1:256] := 0;
```

VDIVPS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] / SRC2[31:0]
DEST[63:32] := SRC1[63:32] / SRC2[63:32]
DEST[95:64] := SRC1[95:64] / SRC2[95:64]
DEST[127:96] := SRC1[127:96] / SRC2[127:96]
DEST[MAXVL-1:128] := 0
```

DIVPS (128-bit Legacy SSE Version)
DEST[31:0] := SRC1[31:0] / SRC2[31:0]
DEST[63:32] := SRC1[63:32] / SRC2[63:32]
DEST[95:64] := SRC1[95:64] / SRC2[95:64]
DEST[127:96] := SRC1[127:96] / SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VDIVPS __m512 _mm512_div_ps(__m512 a, __m512 b);
VDIVPS __m512 _mm512_mask_div_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VDIVPS __m512 _mm512_maskz_div_ps(__mmask16 k, __m512 a, __m512 b);
VDIVPD __m256d _mm256_mask_div_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VDIVPD __m256d _mm256_maskz_div_pd(__mmask8 k, __m256d a, __m256d b);
VDIVPD __m128d _mm_mask_div_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VDIVPD __m128d _mm_maskz_div_pd(__mmask8 k, __m128d a, __m128d b);
VDIVPS __m512 _mm512_div_round_ps(__m512 a, __m512 b, int);
VDIVPS __m512 _mm512_mask_div_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VDIVPS __m512 _mm512_maskz_div_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VDIVPS __m256 _mm256_div_ps (__m256 a, __m256 b);
DIVPS __m128 _mm_div_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, "Type 2 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

DIVSD—Divide Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5E /r DIVSD xmm1, xmm2/m64	A	V/V	SSE2	Divide low double precision floating-point value in xmm1 by low double precision floating-point value in xmm2/m64.
VEX.LIG.F2.0F.WIG 5E /r VDIVSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Divide low double precision floating-point value in xmm2 by low double precision floating-point value in xmm3/m64.
EVEX.LLIG.F2.0F.W1 5E /r VDIVSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F OR AVX10.1	Divide low double precision floating-point value in xmm2 by low double precision floating-point value in xmm3/m64.

Instruction Operand Encoding

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

Description

Divides the low double precision floating-point value in the first source operand by the low double precision floating-point value in the second source operand, and stores the double precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination are XMM registers.

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

VEX.128 encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The quadword at bits 127:64 of the destination operand is copied from the corresponding quadword of the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.128 encoded version: The first source operand is an xmm register encoded by EVEX.vvvv. The quadword element of the destination operand at bits 127:64 are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

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

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

Operation

VDIVSD (EVEX Encoded Version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := SRC1[63:0] / SRC2[63:0]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VDIVSD (VEX.128 Encoded Version)

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

DIVSD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] / SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VDIVSD __m128d _mm_mask_div_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VDIVSD __m128d _mm_maskz_div_sd(__mmask8 k, __m128d a, __m128d b);
VDIVSD __m128d _mm_div_round_sd(__m128d a, __m128d b, int);
VDIVSD __m128d _mm_mask_div_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VDIVSD __m128d _mm_maskz_div_round_sd(__mmask8 k, __m128d a, __m128d b, int);
DIVSD __m128d _mm_div_sd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-49, "Type E3 Class Exception Conditions."

DIVSS—Divide Scalar Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5E /r DIVSS xmm1, xmm2/m32	A	V/V	SSE	Divide low single precision floating-point value in xmm1 by low single precision floating-point value in xmm2/m32.
VEX.LIG.F3.0F.WIG 5E /r VDIVSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Divide low single precision floating-point value in xmm2 by low single precision floating-point value in xmm3/m32.
EVEX.LLIG.F3.0F.W0 5E /r VDIVSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	C	V/V	AVX512F OR AVX10.1	Divide low single precision floating-point value in xmm2 by low single precision floating-point value in xmm3/m32.

Instruction Operand Encoding

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

Description

Divides the low single precision floating-point value in the first source operand by the low single precision floating-point value in the second source operand, and stores the single precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location.

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

VEX.128 encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The three high-order doublewords of the destination operand are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.128 encoded version: The first source operand is an xmm register encoded by EVEX.vvvv. The doubleword elements of the destination operand at bits 127:32 are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

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

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

Operation

VDIVSS (EVEX Encoded Version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := SRC1[31:0] / SRC2[31:0]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VDIVSS (VEX.128 Encoded Version)

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

DIVSS (128-bit Legacy SSE Version)

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

Intel C/C++ Compiler Intrinsic Equivalent

VDIVSS __m128 _mm_mask_div_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VDIVSS __m128 _mm_maskz_div_ss(__mmask8 k, __m128 a, __m128 b);
VDIVSS __m128 _mm_div_round_ss(__m128 a, __m128 b, int);
VDIVSS __m128 _mm_mask_div_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VDIVSS __m128 _mm_maskz_div_round_ss(__mmask8 k, __m128 a, __m128 b, int);
DIVSS __m128 _mm_div_ss(__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-49, "Type E3 Class Exception Conditions."

DPPD—Dot Product of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 41 /r ib DPPD xmm1, xmm2/m128, imm8	RMI	V/V	SSE4_1	Selectively multiply packed double precision floating-point values from xmm1 with packed double precision floating-point values from xmm2, add and selectively store the packed double precision floating-point values to xmm1.
VEX.128.66.0F3A.WIG 41 /r ib VDPPD xmm1,xmm2, xmm3/m128, imm8	RVMI	V/V	AVX	Selectively multiply packed double precision floating-point values from xmm2 with packed double precision floating-point values from xmm3, add and selectively store the packed double precision floating-point values to xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Conditionally multiplies the packed double precision floating-point values in the destination operand (first operand) with the packed double precision floating-point values in the source (second operand) depending on a mask extracted from bits [5:4] of the immediate operand (third operand). If a condition mask bit is zero, the corresponding multiplication is replaced by a value of 0.0 in the manner described by Section 12.8.4 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

The two resulting double precision values are summed into an intermediate result. The intermediate result is conditionally broadcasted to the destination using a broadcast mask specified by bits [1:0] of the immediate byte.

If a broadcast mask bit is "1", the intermediate result is copied to the corresponding qword element in the destination operand. If a broadcast mask bit is zero, the corresponding element in the destination is set to zero.

DPPD follows the NaN forwarding rules stated in the Software Developer's Manual, vol. 1, table 4.7. These rules do not cover horizontal prioritization of NaNs. Horizontal propagation of NaNs to the destination and the positioning of those NaNs in the destination is implementation dependent. NaNs on the input sources or computationally generated NaNs will have at least one NaN propagated to the destination.

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

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

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

Operation

DP_primitive (SRC1, SRC2)

```
IF (imm8[4] = 1)
    THEN Temp1[63:0] := DEST[63:0] * SRC[63:0]; // update SIMD exception flags
    ELSE Temp1[63:0] := +0.0; FI;
IF (imm8[5] = 1)
    THEN Temp1[127:64] := DEST[127:64] * SRC[127:64]; // update SIMD exception flags
    ELSE Temp1[127:64] := +0.0; FI;
/* if unmasked exception reported, execute exception handler*/

Temp2[63:0] := Temp1[63:0] + Temp1[127:64]; // update SIMD exception flags
/* if unmasked exception reported, execute exception handler*/

IF (imm8[0] = 1)
    THEN DEST[63:0] := Temp2[63:0];
    ELSE DEST[63:0] := +0.0; FI;
IF (imm8[1] = 1)
    THEN DEST[127:64] := Temp2[63:0];
    ELSE DEST[127:64] := +0.0; FI;
```

DPPD (128-bit Legacy SSE Version)

```
DEST[127:0] := DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[MAXVL-1:128] (Unmodified)
```

VDPPD (VEX.128 Encoded Version)

```
DEST[127:0] := DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[MAXVL-1:128] := 0
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

DPPD __m128d _mm_dp_pd (__m128d a, __m128d b, const int mask);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Exceptions are determined separately for each add and multiply operation. Unmasked exceptions will leave the destination untouched.

Other Exceptions

See Table 2-19, "Type 2 Class Exception Conditions," additionally:

#UD	If VEX.L= 1.

DPPS—Dot Product of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 40 /r ib DPPS xmm1, xmm2/m128, imm8	RMI	V/V	SSE4_1	Selectively multiply packed single precision floating-point values from xmm1 with packed single precision floating-point values from xmm2, add and selectively store the packed single precision floating-point values or zero values to xmm1.
VEX.128.66.0F3A.WIG 40 /r ib VDPPS xmm1,xmm2, xmm3/m128, imm8	RVMI	V/V	AVX	Multiply packed single precision floating-point values from xmm1 with packed single precision floating-point values from xmm2/mem selectively add and store to xmm1.
VEX.256.66.0F3A.WIG 40 /r ib VDPPS ymm1, ymm2, ymm3/m256, imm8	RVMI	V/V	AVX	Multiply packed single precision floating-point values from ymm2 with packed single precision floating-point values from ymm3/mem, selectively add pairs of elements and store to ymm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Conditionally multiplies the packed single precision floating-point values in the destination operand (first operand) with the packed single precision floats in the source (second operand) depending on a mask extracted from the high 4 bits of the immediate byte (third operand). If a condition mask bit in imm8[7:4] is zero, the corresponding multiplication is replaced by a value of 0.0 in the manner described by Section 12.8.4 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

The four resulting single precision values are summed into an intermediate result. The intermediate result is conditionally broadcasted to the destination using a broadcast mask specified by bits [3:0] of the immediate byte.

If a broadcast mask bit is "1", the intermediate result is copied to the corresponding dword element in the destination operand. If a broadcast mask bit is zero, the corresponding element in the destination is set to zero.

DPPS follows the NaN forwarding rules stated in the Software Developer's Manual, vol. 1, table 4.7. These rules do not cover horizontal prioritization of NaNs. Horizontal propagation of NaNs to the destination and the positioning of those NaNs in the destination is implementation dependent. NaNs on the input sources or computationally generated NaNs will have at least one NaN propagated to the destination.

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

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

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

Operation

DP_primitive (SRC1, SRC2)

```
IF (imm8[4] = 1)
    THEN Temp1[31:0] := DEST[31:0] * SRC[31:0]; // update SIMD exception flags
    ELSE Temp1[31:0] := +0.0; FI;
IF (imm8[5] = 1)
    THEN Temp1[63:32] := DEST[63:32] * SRC[63:32]; // update SIMD exception flags
    ELSE Temp1[63:32] := +0.0; FI;
IF (imm8[6] = 1)
    THEN Temp1[95:64] := DEST[95:64] * SRC[95:64]; // update SIMD exception flags
    ELSE Temp1[95:64] := +0.0; FI;
IF (imm8[7] = 1)
    THEN Temp1[127:96] := DEST[127:96] * SRC[127:96]; // update SIMD exception flags
    ELSE Temp1[127:96] := +0.0; FI;

Temp2[31:0] := Temp1[31:0] + Temp1[63:32]; // update SIMD exception flags
/* if unmasked exception reported, execute exception handler*/
Temp3[31:0] := Temp1[95:64] + Temp1[127:96]; // update SIMD exception flags
/* if unmasked exception reported, execute exception handler*/
Temp4[31:0] := Temp2[31:0] + Temp3[31:0]; // update SIMD exception flags
/* if unmasked exception reported, execute exception handler*/

IF (imm8[0] = 1)
    THEN DEST[31:0] := Temp4[31:0];
    ELSE DEST[31:0] := +0.0; FI;
IF (imm8[1] = 1)
    THEN DEST[63:32] := Temp4[31:0];
    ELSE DEST[63:32] := +0.0; FI;
IF (imm8[2] = 1)
    THEN DEST[95:64] := Temp4[31:0];
    ELSE DEST[95:64] := +0.0; FI;
IF (imm8[3] = 1)
    THEN DEST[127:96] := Temp4[31:0];
    ELSE DEST[127:96] := +0.0; FI;
```

DPPS (128-bit Legacy SSE Version)

```
DEST[127:0] := DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[MAXVL-1:128] (Unmodified)
```

VDPPS (VEX.128 Encoded Version)

```
DEST[127:0] := DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[MAXVL-1:128] := 0
```

VDPPS (VEX.256 Encoded Version)

```
DEST[127:0] := DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[255:128] := DP_Primitive(SRC1[255:128], SRC2[255:128]);
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

(V)DPPS __m128 _mm_dp_ps (__m128 a, __m128 b, const int mask);
VDPPS __m256 _mm256_dp_ps (__m256 a, __m256 b, const int mask);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Exceptions are determined separately for each add and multiply operation, in the order of their execution. Unmasked exceptions will leave the destination operands unchanged.

Other Exceptions

See Table 2-19, "Type 2 Class Exception Conditions."

EMMS—Empty MMX Technology State

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F 77	EMMS	ZO	Valid	Valid	Set the x87 FPU tag word to empty.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Sets the values of all the tags in the x87 FPU tag word to empty (all 1s). This operation marks the x87 FPU data registers (which are aliased to the MMX technology registers) as available for use by x87 FPU floating-point instructions. (See Figure 8-7 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for the format of the x87 FPU tag word.) All other MMX instructions (other than the EMMS instruction) set all the tags in x87 FPU tag word to valid (all 0s).

The EMMS instruction must be used to clear the MMX technology state at the end of all MMX technology procedures or subroutines and before calling other procedures or subroutines that may execute x87 floating-point instructions. If a floating-point instruction loads one of the registers in the x87 FPU data register stack before the x87 FPU tag word has been reset by the EMMS instruction, an x87 floating-point register stack overflow can occur that will result in an x87 floating-point exception or incorrect result.

EMMS operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
x87FPUTagWord := FFFFH;
```

Intel C/C++ Compiler Intrinsic Equivalent

void _mm_empty()

Flags Affected

None

Protected Mode Exceptions

#UD	If CR0.EM[bit 2] = 1.
#NM	If CR0.TS[bit 3] = 1.
#MF	If there is a pending FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

ENCODEKEY128—Encode 128-Bit Key With Key Locker

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 38 FA 11:rrr:bbb ENCODEKEY128 r32, r32, <XMM0-2>, <XMM4-6>	A	V/V	AESKLE	Wrap a 128-bit AES key from XMM0 into a key handle and output handle in XMM0—2.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operands 4—5	Operands 6—7
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	Implicit XMM0 (r, w)	Implicit XMM1—2 (w)	Implicit XMM4—6 (w)

Description

The ENCODEKEY128[1] instruction wraps a 128-bit AES key from the implicit operand XMM0 into a key handle that is then stored in the implicit destination operands XMM0-2.

The explicit source operand specifies handle restrictions, if any.

The explicit destination operand is populated with information on the source of the key and its attributes. XMM4 through XMM6 are reserved for future usages and software should not rely upon them being zeroed.

Operation

ENCODEKEY128

```
#GP (0) if a reserved bit[2] in SRC[31:0] is set
InputKey[127:0] := XMM0;
KeyMetadata[2:0] = SRC[2:0];
KeyMetadata[23:3] = 0;          // Reserved for future usage
KeyMetadata[27:24] = 0;         // KeyType is AES-128 (value of 0)
KeyMetadata[127:28] = 0;        // Reserved for future usage

// KeyMetadata is the AAD input and InputKey is the Plaintext input for WrapKey128
Handle[383:0] := WrapKey128(InputKey[127:0], KeyMetadata[127:0], IWKey.Integrity Key[127:0], IWKey.Encryption Key[255:0]);

DEST[0] := IWKey.NoBackup;
DEST[4:1] := IWKey.KeySource[3:0];
DEST[31:5] = 0;
XMM0 := Handle[127:0];     // AAD
XMM1 := Handle[255:128]; // Integrity Tag
XMM2 := Handle[383:256]; // CipherText
XMM4 := 0;   // Reserved for future usage
XMM5 := 0;   // Reserved for future usage
XMM6 := 0;   // Reserved for future usage
RFLAGS.OF, SF, ZF, AF, PF, CF := 0;
```

Flags Affected

All arithmetic flags (OF, SF, ZF, AF, PF, CF) are cleared to 0. Although they are cleared for the currently defined operations, future extensions may report information in the flags.

---

1. Further details on Key Locker and usage of this instruction can be found here: https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html.
2. SRC[31:3] are currently reserved for future usages. SRC[2], which indicates a no-decrypt restriction, is reserved if CPUID.19H:EAX[2] is 0. SRC[1], which indicates a no-encrypt restriction, is reserved if CPUID.19H:EAX[1] is 0. SRC[0], which indicates a CPL0-only restriction, is reserved if CPUID.19H:EAX[0] is 0.

Intel C/C++ Compiler Intrinsic Equivalent

ENCODEKEY128 unsigned int _mm_encodekey128_u32(unsigned int htype, __m128i key, void* h);

Exceptions (All Operating Modes)

#GP	If reserved bit is set in source register value.
#UD	If the LOCK prefix is used.
	If CPUID.07H.00H:ECX.KEY_LOCKER[23] = 0.
	If CR4.KL = 0.
	If CPUID.19H:EBX.AESKLE[0] = 0.
	If CR0.EM = 1.
	If CR4.OSFXSR = 0.
#NM	If CR0.TS = 1.

ENCODEKEY256—Encode 256-Bit Key With Key Locker

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 38 FB 11:rrr:bbb ENCODEKEY256 r32, r32 <XMM0-6>	A	V/V	AESKLE	Wrap a 256-bit AES key from XMM1:XMM0 into a key handle and store it in XMM0—3.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operands 3—4	Operands 5—9
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	Implicit XMM0—1 (r, w)	Implicit XMM2—6 (w)

Description

The ENCODEKEY256[1] instruction wraps a 256-bit AES key from the implicit operand XMM1:XMM0 into a key handle that is then stored in the implicit destination operands XMM0-3.

The explicit source operand is a general-purpose register and specifies what handle restrictions should be built into the handle.

The explicit destination operand is populated with information on the source of the key and its attributes. XMM4 through XMM6 are reserved for future usages and software should not rely upon them being zeroed.

Operation

ENCODEKEY256

```
#GP (0) if a reserved bit[2] in SRC[31:0] is set
InputKey[255:0] := XMM1:XMM0;
KeyMetadata[2:0] = SRC[2:0];
KeyMetadata[23:3] = 0;      // Reserved for future usage
KeyMetadata[27:24] = 1;    // KeyType is AES-256 (value of 1)
KeyMetadata[127:28] = 0; // Reserved for future usage

// KeyMetadata is the AAD input and InputKey is the Plaintext input for WrapKey256
Handle[511:0] := WrapKey256(InputKey[255:0], KeyMetadata[127:0], IWKey.Integrity Key[127:0], IWKey.Encryption Key[255:0]);

DEST[0] := IWKey.NoBackup;
DEST[4:1] := IWKey.KeySource[3:0];
DEST[31:5] = 0;
XMM0 := Handle[127:0];      // AAD
XMM1 := Handle[255:128]; // Integrity Tag
XMM2 := Handle[383:256]; // CipherText[127:0]
XMM3 := Handle[511:384]; // CipherText[255:128]

XMM4 := 0;   // Reserved for future usage
XMM5 := 0;   // Reserved for future usage
XMM6 := 0;   // Reserved for future usage

RFLAGS.OF, SF, ZF, AF, PF, CF := 0;
```

1. Further details on Key Locker and usage of this instruction can be found here: https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html.
2. SRC[31:3] are currently reserved for future usages. SRC[2], which indicates a no-decrypt restriction, is reserved if CPUID.19H:EAX[2] is 0. SRC[1], which indicates a no-encrypt restriction, is reserved if CPUID.19H:EAX[1] is 0. SRC[0], which indicates a CPL0-only restriction, is reserved if CPUID.19H:EAX[0] is 0.

Flags Affected

All arithmetic flags (OF, SF, ZF, AF, PF, CF) are cleared to 0. Although they are cleared for the currently defined operations, future extensions may report information in the flags.

Intel C/C++ Compiler Intrinsic Equivalent

ENCODEKEY256 unsigned int _mm_encodekey256_u32(unsigned int htype, __m128i key_lo, __m128i key_hi, void* h);

Exceptions (All Operating Modes)

#GP	If reserved bit is set in source register value.
#UD	If the LOCK prefix is used.
	If CPUID.07H.00H:ECX.KEY_LOCKER[23] = 0.
	If CR4.KL = 0.
	If CPUID.19H:EBX.AESKLE[0] = 0.
	If CR0.EM = 1.
	If CR4.OSFXSR = 0.
#NM	If CR0.TS = 1.

ENDBR32—Terminate an Indirect Branch in 32-bit and Compatibility Mode

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 1E FB ENDBR32	ZO	V/V	CET_IBT	Terminate indirect branch in 32-bit and compatibility mode.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A	N/A

Description

Terminate an indirect branch in 32 bit and compatibility mode. This opcode is a NOP when CET indirect branch tracking is not enabled, on processors that do not support CET, and in 64-bit mode.

Operation

```
IF EndbranchEnabled(CPL) & (IA32_EFER.LMA = 0 | (IA32_EFER.LMA=1 & CS.L = 0)
    IF CPL = 3
        THEN
            IA32_U_CET.TRACKER = IDLE
            IA32_U_CET.SUPPRESS = 0
        ELSE
            IA32_S_CET.TRACKER = IDLE
            IA32_S_CET.SUPPRESS = 0
    FI;
FI;
```

Flags Affected

None.

Exceptions

#UD	If the LOCK prefix is used.

ENDBR64—Terminate an Indirect Branch in 64-bit Mode

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 1E FA ENDBR64	ZO	V/V	CET_IBT	Terminate indirect branch in 64-bit mode.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A	N/A

Description

Terminate an indirect branch in 64 bit mode. This opcode is a NOP when CET indirect branch tracking is not enabled, on processors that do not support CET, and outside 64-bit mode.

Operation

```
IF EndbranchEnabled(CPL) & IA32_EFER.LMA = 1 & CS.L = 1
    IF CPL = 3
        THEN
            IA32_U_CET.TRACKER = IDLE
            IA32_U_CET.SUPPRESS = 0
        ELSE
            IA32_S_CET.TRACKER = IDLE
            IA32_S_CET.SUPPRESS = 0
    FI;
FI;
```

Flags Affected

None.

Exceptions

#UD	If the LOCK prefix is used.

ENQCMD—Enqueue Command

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 38 F8 !(11):rrr:bbb ENQCMD r32/r64, m512	A	V/V	ENQCMD	Atomically enqueue 64-byte user command from source memory operand to destination offset in ES segment specified in register operand as offset in ES segment.

Instruction Operand Encoding

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

Description

The ENQCMD instruction allows software to write commands to **enqueue registers**, which are special device registers accessed using memory-mapped I/O (MMIO).

Enqueue registers expect writes to have the following format:

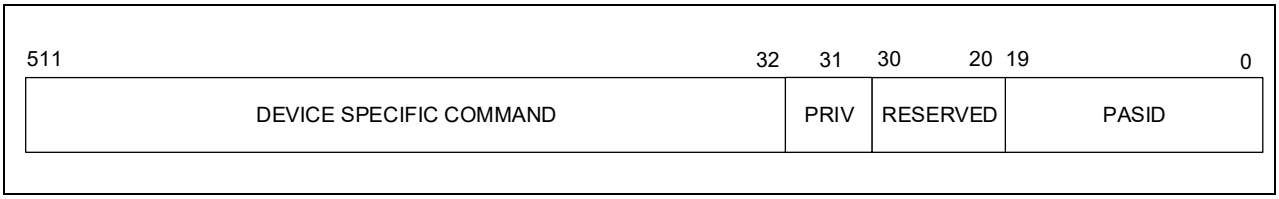


Figure 3-11. 64-Byte Data Written to Enqueue Registers

Bits 19:0 convey the process address space identifier (PASID), a value which system software may assign to individual software threads. Bit 31 contains privilege identification (0 = user; 1 = supervisor). Devices implementing enqueue registers may use these two values along with a device-specific command in the upper 60 bytes.

The ENQCMD instruction begins by reading 64 bytes of command data from its source memory operand. This is an ordinary load with cacheability and memory ordering implied normally by the memory type. The source operand need not be aligned, and there is no guarantee that all 64 bytes are loaded atomically. Bits 31:0 of the source operand must be zero.

The instruction then formats those 64 bytes into **command data** with a format consistent with that given in Figure 3-11:

- Command[19:0] get IA32_PASID[19:0].[1]
- Command[30:20] are zero.
- Command[31] is 0 (indicating user; this value is used regardless of CPL).
- Command[511:32] get bits 511:32 of the source operand that was read from memory.

The ENQCMD instruction uses an **enqueue store** (defined below) to write this command data to the destination operand. The address of the destination operand is specified in a general-purpose register as an offset into the ES segment (the segment cannot be overridden).[2] The destination linear address must be 64-byte aligned. The operation of an enqueue store disregards the memory type of the destination memory address.

1. It is expected that system software will load the IA32_PASID MSR so that bits 19:0 contain the PASID of the current software thread. The MSR's valid bit, IA32_PASID[31], must be 1. For additional details on the IA32_PASID MSR, see the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4.
2. In 64-bit mode, the width of the register operand is 64 bits (32 bits with a 67H prefix). Outside 64-bit mode when CS.D = 1, the width is 32 bits (16 bits with a 67H prefix). Outside 64-bit mode when CS.D=0, the width is 16 bits (32 bits with a 67H prefix).

An enqueue store is not ordered relative to older stores to WB or WC memory (including non-temporal stores) or to executions of the CLFLUSHOPT or CLWB (when applied to addresses other than that of the enqueue store). Software can enforce such ordering by executing a fencing instruction such as SFENCE or MFENCE before the enqueue store.

An enqueue store does not write the data into the cache hierarchy, nor does it fetch any data into the cache hierarchy. An enqueue store's command data is never combined with that of any other store to the same address.

Unlike other stores, an enqueue store returns a status, which the ENQCMD instruction loads into the ZF flag in the RFLAGS register:

- ZF = 0 (success) reports that the 64-byte command data was written atomically to a device's enqueue register and has been accepted by the device. (It does not guarantee that the device has acted on the command; it may have queued it for later execution.)
- ZF = 1 (retry) reports that the command data was not accepted. This status is returned if the destination address is an enqueue register but the command was not accepted due to capacity or other temporal reasons. This status is also returned if the destination address was not an enqueue register (including the case of a memory address); in these cases, the store is dropped and is written neither to MMIO nor to memory.

Availability of the ENQCMD instruction is indicated by the presence of the CPUID feature flag ENQCMD (CPUID.07H.00H:ECX[29]).

Operation

```
IF IA32_PASID[31] = 0
    THEN #GP;
ELSE
    COMMAND := (SRC & ~FFFFFFFFH) | (IA32_PASID & FFFFFH);
    DEST := COMMAND;
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

ENQCMD int_enqcmd(void *dst, const void *src)

Flags Affected

The ZF flag is set if the enqueue-store completion returns the retry status; otherwise it is cleared. All other flags are cleared.

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If destination linear address is not aligned to a 64-byte boundary.
	If the PASID Valid field (bit 31) is 0 in IA32_PASID MSR.
	If bits 31:0 of the source operand are not all zero.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#UD	If CPUID.07H.00H:ECX.ENQCMD[29] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
	If destination linear address is not aligned to a 64-byte boundary.
	If the PASID Valid field (bit 31) is 0 in IA32_PASID MSR.
	If bits 31:0 of the source operand are not all zero.
#UD	If CPUID.07H.00H:ECX.ENQCMD[29] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real-address mode. Additionally:

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in non-canonical form.
#GP(0)	If the memory address is in non-canonical form.
	If destination linear address is not aligned to a 64-byte boundary.
	If the PASID Valid field (bit 31) is 0 in IA32_PASID MSR.
	If bits 31:0 of the source operand are not all zero.
#PF(fault-code)	For a page fault.
#UD	If CPUID.07H.00H:ECX.ENQCMD[29].
	If the LOCK prefix is used.

ENQCMDS—Enqueue Command Supervisor

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 38 F8 !(11):rrr:bbb ENQCMDS r32/r64, m512	A	V/V	ENQCMD	Atomically enqueue 64-byte command with PASID from source memory operand to destination offset in ES segment specified in register operand as offset in ES segment.

Instruction Operand Encoding

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

Description

The ENQCMDS instruction allows system software to write commands to **enqueue registers**, which are special device registers accessed using memory-mapped I/O (MMIO).

Enqueue registers expect writes to have the format given in Figure 3-11 and explained in the section on "ENQCMD—Enqueue Command."

The ENQCMDS instruction begins by reading 64 bytes of command data from its source memory operand. This is an ordinary load with cacheability and memory ordering implied normally by the memory type. The source operand need not be aligned, and there is no guarantee that all 64 bytes are loaded atomically. Bits 30:20 of the source operand must be zero.

ENQCMDS formats its source data differently from ENQCMD. Specifically, it formats them into **command data** as follows:

- Command[19:0] get bits 19:0 of the source operand that was read from memory. These 20 bits communicate a process address-space identifier (PASID).
- Command[30:20] are zero.
- Command[511:31] get bits 511:31 of the source operand that was read from memory. Bit 31 communicates a privilege identification (0 = user; 1 = supervisor).

The ENQCMDS instruction then uses an **enqueue store** (defined below) to write this command data to the destination operand. The address of the destination operand is specified in a general-purpose register as an offset into the ES segment (the segment cannot be overridden).[1] The destination linear address must be 64-byte aligned. The operation of an enqueue store disregards the memory type of the destination memory address.

An enqueue store is not ordered relative to older stores to WB or WC memory (including non-temporal stores) or to executions of the CLFLUSHOPT or CLWB (when applied to addresses other than that of the enqueue store). Software can enforce such ordering by executing a fencing instruction such as SFENCE or MFENCE before the enqueue store.

An enqueue store does not write the data into the cache hierarchy, nor does it fetch any data into the cache hierarchy. An enqueue store's command data is never combined with that of any other store to the same address.

Unlike other stores, an enqueue store returns a status, which the ENQCMDS instruction loads into the ZF flag in the RFLAGS register:

- ZF = 0 (success) reports that the 64-byte command data was written atomically to a device's enqueue register and has been accepted by the device. (It does not guarantee that the device has acted on the command; it may have queued it for later execution.)
- ZF = 1 (retry) reports that the command data was not accepted. This status is returned if the destination address is an enqueue register but the command was not accepted due to capacity or other temporal reasons.

1. In 64-bit mode, the width of the register operand is 64 bits (32 bits with a 67H prefix). Outside 64-bit mode when CS.D = 1, the width is 32 bits (16 bits with a 67H prefix). Outside 64-bit mode when CS.D=0, the width is 16 bits (32 bits with a 67H prefix).

This status is also returned if the destination address was not an enqueue register (including the case of a memory address); in these cases, the store is dropped and is written neither to MMIO nor to memory.

The ENQCMDS instruction may be executed only if CPL= 0. Availability of the ENQCMDS instruction is indicated by the presence of the CPUID feature flag ENQCMD (CPUID.07H.00H:ECX[29]).

Operation

```
DEST := SRC;
```

Intel C/C++ Compiler Intrinsic Equivalent

ENQCMDS int_enqcmds(void *dst, const void *src)

Flags Affected

The ZF flag is set if the enqueue-store completion returns the retry status; otherwise it is cleared. All other flags are cleared.

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If destination linear address is not aligned to a 64-byte boundary.
	If the current privilege level is not 0.
	If bits 30:20 of the source operand are not all zero.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#UD	If CPUID.07H.00H:ECX.ENQCMD[29] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
	If destination linear address is not aligned to a 64-byte boundary.
	If bits 30:20 of the source operand are not all zero.
#UD	If CPUID.07H.00H:ECX.ENQCMD[29] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	The ENQCMDS instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in non-canonical form.
#GP(0)	If the memory address is in non-canonical form.
	If destination linear address is not aligned to a 64-byte boundary.
	If the current privilege level is not 0.
	If bits 30:20 of the source operand are not all zero.
#PF(fault-code)	For a page fault.

#UD	If CPUID.07H.00H:ECX.ENQCMD[29]. If the LOCK prefix is used.

ENTER—Make Stack Frame for Procedure Parameters

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
C8 iw 00	ENTER imm16, 0	II	Valid	Valid	Create a stack frame for a procedure.
C8 iw 01	ENTER imm16,1	II	Valid	Valid	Create a stack frame with a nested pointer for a procedure.
C8 iw ib	ENTER imm16, imm8	II	Valid	Valid	Create a stack frame with nested pointers for a procedure.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
II	iw	imm8	N/A	N/A

Description

Creates a stack frame (comprising of space for dynamic storage and 1-32 frame pointer storage) for a procedure. The first operand (imm16) specifies the size of the dynamic storage in the stack frame (that is, the number of bytes of dynamically allocated on the stack for the procedure). The second operand (imm8) gives the lexical nesting level (0 to 31) of the procedure. The nesting level (imm8 mod 32) and the OperandSize attribute determine the size in bytes of the storage space for frame pointers.

The nesting level determines the number of frame pointers that are copied into the "display area" of the new stack frame from the preceding frame. The default size of the frame pointer is the StackAddrSize attribute, but can be overridden using the 66H prefix. Thus, the OperandSize attribute determines the size of each frame pointer that will be copied into the stack frame and the data being transferred from SP/ESP/RSP register into the BP/EBP/RBP register.

The ENTER and companion LEAVE instructions are provided to support block structured languages. The ENTER instruction (when used) is typically the first instruction in a procedure and is used to set up a new stack frame for a procedure. The LEAVE instruction is then used at the end of the procedure (just before the RET instruction) to release the stack frame.

If the nesting level is 0, the processor pushes the frame pointer from the BP/EBP/RBP register onto the stack, copies the current stack pointer from the SP/ESP/RSP register into the BP/EBP/RBP register, and loads the SP/ESP/RSP register with the current stack-pointer value minus the value in the size operand. For nesting levels of 1 or greater, the processor pushes additional frame pointers on the stack before adjusting the stack pointer. These additional frame pointers provide the called procedure with access points to other nested frames on the stack. See "Procedure Calls for Block-Structured Languages" in Chapter 6 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for more information about the actions of the ENTER instruction.

The ENTER instruction causes a page fault whenever a write using the final value of the stack pointer (within the current stack segment) would do so.

In 64-bit mode, default operation size is 64 bits; 32-bit operation size cannot be encoded. Use of 66H prefix changes frame pointer operand size to 16 bits.

When the 66H prefix is used and causing the OperandSize attribute to be less than the StackAddrSize, software is responsible for the following:

- The companion LEAVE instruction must also use the 66H prefix,
- The value in the RBP/EBP register prior to executing "66H ENTER" must be within the same 16KByte region of the current stack pointer (RSP/ESP), such that the value of RBP/EBP after "66H ENTER" remains a valid address in the stack. This ensures "66H LEAVE" can restore 16-bits of data from the stack.

Operation

```
AllocSize := imm16;
NestingLevel := imm8 MOD 32;
IF (OperandSize = 64)
    THEN
        Push(RBP); (* RSP decrements by 8 *)
        FrameTemp := RSP;
    ELSE IF OperandSize = 32
        THEN
            Push(EBP); (* (E)SP decrements by 4 *)
            FrameTemp := ESP; FI;
    ELSE (* OperandSize = 16 *)
            Push(BP); (* RSP or (E)SP decrements by 2 *)
            FrameTemp := SP;
FI;

IF NestingLevel = 0
    THEN GOTO CONTINUE;
FI;
IF (NestingLevel > 1)
    THEN FOR i := 1 to (NestingLevel - 1)
        DO
            IF (OperandSize = 64)
                THEN
                    RBP := RBP - 8;
                    Push([RBP]); (* Quadword push *)
                ELSE IF OperandSize = 32
                    THEN
                        IF StackSize = 32
                            EBP := EBP - 4;
                            Push([EBP]); (* Doubleword push *)
                        ELSE (* StackSize = 16 *)
                            BP := BP - 4;
                            Push([BP]); (* Doubleword push *)
                        FI;
                    FI;
                ELSE (* OperandSize = 16 *)
                    IF StackSize = 64
                        THEN
                            RBP := RBP - 2;
                            Push([RBP]); (* Word push *)
                    ELSE IF StackSize = 32
                        THEN
                            EBP := EBP - 2;
                            Push([EBP]); (* Word push *)
                        ELSE (* StackSize = 16 *)
                            BP := BP - 2;
                            Push([BP]); (* Word push *)
                    FI;
                FI;
    OD;
FI;
IF (OperandSize = 64) (* nestinglevel 1 *)
```

```
    THEN
        Push(FrameTemp); (* Quadword push and RSP decrements by 8 *)
    ELSE IF OperandSize = 32
        THEN
            Push(FrameTemp); FI; (* Doubleword push and (E)SP decrements by 4 *)
    ELSE (* OperandSize = 16 *)
            Push(FrameTemp); (* Word push and RSP|ESP|SP decrements by 2 *)
FI;
CONTINUE:
IF 64-Bit Mode (StackSize = 64)
    THEN
            RBP := FrameTemp;
            RSP := RSP – AllocSize;
    ELSE IF OperandSize = 32
        THEN
            EBP := FrameTemp;
            ESP := ESP – AllocSize; FI;
    ELSE (* OperandSize = 16 *)
            BP := FrameTemp[15:1]; (* Bits 16 and above of applicable RBP/EBP are unmodified *)
            SP := SP – AllocSize;
FI;
END;
```

Flags Affected

None.

Protected Mode Exceptions

#SS(0)	If the new value of the SP or ESP register is outside the stack segment limit.
#PF(fault-code)	If a page fault occurs or if a write using the final value of the stack pointer (within the current stack segment) would cause a page fault.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#SS	If the new value of the SP or ESP register is outside the stack segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#SS(0)	If the new value of the SP or ESP register is outside the stack segment limit.
#PF(fault-code)	If a page fault occurs or if a write using the final value of the stack pointer (within the current stack segment) would cause a page fault.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs or if a write using the final value of the stack pointer (within the current stack segment) would cause a page fault.
#UD	If the LOCK prefix is used.

ERETS—Event Return to Supervisor

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F2 0F 01 CA	ERETS	ZO	Valid	Invalid	Event return for events occurring in ring 0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

ERETS returns from an event handler, establishing the state based on the contents of the stack (typically, that which was in effect before FRED event delivery). ERETS can be executed only if CPL = 0, and it does not change CPL. For this reason, ERETS is used to return from handling events that occurred while CPL = 0.

ERETS takes no explicit arguments; its operation depends on the contents of the regular stack and (when enabled) the shadow stack.

Execution of ERETS causes an invalid-opcode exception (#UD) if FRED transitions are not enabled or if CPL > 0. For this reason, ERETS can be executed only in 64-bit mode.

Section 8.4.1, "ERETS (Event Return to Supervisor)," of the Intel® 64 and IA-32 Architectures software developer's Manual, Volume 3 includes a detailed discussion of ERETS.

Instruction ordering. Instructions following execution of ERETS may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to ERETS have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

Operation

```
IF CR4.FRED = 0 OR CPL > 0
    THEN #UD;
FI;
// CR4.FRED = 1 and CPL = 0 implies IA32_EFER.LMA = CS.L = 1

// pop old state from regular stack and check it
RSP := RSP + 8;             // skip over error code so that RSP references the return state
pop8B newRIP;
pop8B tempCS;               // not used to load CS
pop8B newRFLAGS;
pop8B newRSP;
pop8B tempSS;               // not used to load SS
IF newRIP is not paging canonical OR
    tempCS & FFFFFFFF_FFF8FFFFH ≠ current CS selector OR
    newRFLAGS & FFFFFFFF_FFC2802AH ≠ 2 OR          // enforce bit 1 set; VM, reserved bits clear
    tempSS & FFF8FFFFH ≠ current SS selector OR    // do not check bits 63:32
        THEN #GP(0);
FI;

// ERETS will not numerically increase stack level
newCSL := min{CSL,tempCS[17:16]};
IBT_restore := tempCS[18];
STI_block := tempSS[16];
pend_DB := tempSS[17];
NMI_unblock := tempSS[18];
```

```
// If supervisor shadow stacks are enabled, pop and check values from the shadow stack
IF CR4.CET = 1 AND IA32_S_CET.SH_STK_EN = 1
    THEN
        IF SSP & 7 ≠ 0                                  // require 8-byte alignment
            THEN #CP(FAR-RET/IRET);
        FI;
        popSS_8B newSSP;
        popSS_8B checkSSLIP;
        popSS_8B checkSSCS;
        IF checkSSCS ≠ tempCS                           // 64-bit compare
            OR checkSSLIP ≠ newRIP
            OR newSSP & 3H ≠ 0
            THEN #CP(FAR-RET/IRET);
        FI;
        IF newSSP not CPU canonical
            THEN #GP(0);
        FI;
        // If the stack level is changing, compare SSP to the FRED SSP MSR for the old stack level
        IF newCSL < CSL AND IA32_FRED_SSPi ≠ SSP    // where i = CSL
            THEN #CP(FAR-RET/IRET);
        FI;
FI;

// update registers for return state
RIP := newRIP;
RFLAGS := newRFLAGS;                                    // ERETS can set RFLAGS.RF to 1
RSP := newRSP;
CSL := newCSL;                                          // reflect in IA32_FRED_CONFIG[1:0]
IF CR4.CET = 1 AND IA32_S_CET.SH_STK_EN = 1
    THEN SSP := newSSP;
FI;
IF CR4.CET = 1 AND IA32_S_CET.ENDBR_EN = 1 AND IA32_S_CET.SUPPRESS = 0 AND IBT_restore = 1
    THEN IA32_S_CET.TRACKER := 1;
FI;

// update event-related state
IF STI_block = 1 AND RFLAGS.IF = 1 AND STI blocking was not in effect prior to ERETS
    THEN establish STI blocking after ERETS;
FI;
IF pend_DB = 1 AND RFLAGS.TF =1
    THEN pend a single-step debug exception (#DB) to be delivered after ERETS;
FI;
IF NMI_unblock = 1
    THEN unblock NMIs;
FI;
```

Flags Affected

All defined flags and fields in the RFLAGS register are potentially modified except for the VM flag.

Protected Mode Exceptions

#UD	The ERETS instruction is not recognized in protected mode.

Real-Address Mode Exceptions

#UD	The ERETS instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The ERETS instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	The ERETS instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD	If CR4.FRED = 0.
	If CPL > 0.
	If the LOCK prefix is used.
#GP(0)	If the RIP image on the stack is a not paging canonical.
	If the RFLAGS image on the stack sets reserved bits or the VM bit (or fails to set bit 1).
	If bits 15:0 of the CS image on the stack differs from the value of the current CS selector.
	If the CS image on the stack sets undefined bits (bits 63:19).
	If bits 15:0 of the SS image on the stack differs from the value of the current SS selector.
	If the SS image on the stack sets undefined bits (bits 31:19).
	If the SSP image on the shadow stack is not CPU canonical.
	If a shadow-stack access would use an address that is not paging canonical or would cause a LASS violation.
#SS(0)	If an ordinary stack access would use an address that is not paging canonical or would cause a LASS violation.
#CP(FAR-RET/IRET)	If the address in SSP is not 8-byte aligned.
	If the 64-bit CS image on the shadow stack differs from that on the regular stack.
	If the SSLIP image on the shadow stack differs from the RIP image on the regular stack.
	If the SSP image on the shadow stack is not 4-byte aligned.
	If the CSL is i, the stack level is changing, and SSP differs from the value of IA32_FRED_SSPi.

ERETU—Event Return to User

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F 01 CA	ERETU	ZO	Valid	Valid	Event return for events occurring in ring 3.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

ERETU returns from an event handler, establishing the state based on the contents of the stack (typically, that which was in effect before FRED event delivery). ERETU can be executed only if CPL = 0, and it changes CPL to 3. For this reason, ERETU is used to return from handling events that occurred while CPL = 3.

ERETU takes no explicit arguments; its operation depends on the contents of the regular stack.

Execution of ERETU causes an invalid-opcode exception (#UD) if FRED transitions are not enabled or if CPL > 0. For this reason, ERETU can be executed only in 64-bit mode.

ERETU establishes new values of CS and SS in one of three different ways:

- If the values popped from the stack correspond to the values of IA32_STAR[63:48] + 16 and IA32_STAR[63:48] + 8, respectively, CS and SS are loaded with standard values for operation in 64-bit mode (similar those established by the 64-bit form of SYSCALL).
- If the values popped from the stack correspond to the values of IA32_STAR[63:48] and IA32_STAR[63:48] + 8, respectively, CS and SS are loaded with standard values for operation in compatibility mode (similar those established by the 32-bit form of SYSCALL).
- Otherwise, CS and SS are loaded from the GDT or LDT using the selectors popped from the stack (similar to the manner used by the 64-bit form of IRET).

For further details see the Operation section below and Section 8.4.2, "ERETU (Event Return to User)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3.

Instruction ordering. Instructions following execution of ERETU may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to ERETU have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

Operation

```
IF CR4.FRED = 0 OR CS.L = 0 OR CPL > 0
    THEN #UD;
FI;
IF CSL > 0
    THEN #GP(0);
FI;
// pop old state from regular stack and check it
RSP := RSP + 8;              // skip over error code so that RSP references the return state
pop8B newRIP;
pop8B tempCS;
pop8B newRFLAGS;
pop8B newRSP;
pop8B tempSS;
IF tempCS & FFFFFFFF_FFFF0003H ≠ 3 OR          // enforce return to ring 3
    newRFLAGS & FFFFFFFF_FFC2B02AH ≠ 2 OR  // enforce bit 1 set; IOPL, VM, reserved bits clear
    tempSS & FFF80003H ≠ 3                         // do not check bits 63:32
        THEN #GP(0);
```

```
FI;
pend_DB := tempSS[17];
NMI_unblock := tempSS[18];
IF tempCS[15:0] = IA32_STAR[63:48] + 16 AND tempSS[15:0] = IA32_STAR[63:48] + 8
    THEN                                        // Return to ring 3 in standard 64-bit configuration
        // set newCS to standard values used ring 3 in 64-bit mode
            newCS.selector := tempCS[15:0];
            newCS.base := 0;
            newCS.limit := FFFFFH;
            newCS.type := 11;
            newCS.S := 1;
            newCS.DPL := 3;
            newCS.P := 1;
            newCS.L := 1;
            newCS.D := 0;
            newCS.G := 1;
            newCS.unusable := 0;
        // set newSS to standard values for ring 3
            newSS.selector := tempSS[15:0];
            newSS.base := 0;
            newSS.limit := FFFFFH;
            newSS.type := 3;
            newSS.S := 1;
            newSS.DPL := 3;
            newSS.P := 1;
            newSS.B := 1;
            newSS.G := 1;
            newSS.unusable := 0;
ELSIF tempCS[15:0] = IA32_STAR[63:48] AND tempSS[15:0] = IA32_STAR[63:48] + 8
    THEN
        // set newCS to standard values used ring 3 in compatibility mode
            newCS.selector := tempCS[15:0];
            newCS.base := 0;
            newCS.limit := FFFFFH;
            newCS.type := 11;
            newCS.S := 1;
            newCS.DPL := 3;
            newCS.P := 1;
            newCS.L := 0;
            newCS.D := 1;
            newCS.G := 1;
            newCS.unusable := 0;
        // set newSS to standard values for ring 3
            newSS.selector := tempSS[15:0];
            newSS.base := 0;
            newSS.limit := FFFFFH;
            newSS.type := 3;
            newSS.S := 1;
            newSS.DPL := 3;
            newSS.P := 1;
            newSS.B := 1;
            newSS.G := 1;
            newSS.unusable := 0;
    ELSE
```

```
        load newCS using tempCS[15:0];          // load each as is done by IRET, including
        load newSS using tempSS[15:0];          // checks that may lead to a fault
FI;
IF newCS.L = 1
    THEN    // return to 64-bit mode
        IF newRIP is not paging canonical
            THEN #GP(0);
        FI;
    ELSE    // return to compatibility mode
        newRIP[63:32] := 0;
        IF newRIP is not within newCS's limit (based on limit field and G bit)
            // newRIP is always within the limit with standard values for ring 3 in compatibility mode
            THEN #GP(0);
        FI;
        newRSP[63:32] := 0;
FI;
// If user shadow stacks are enabled, check new SSP value on return to compatibility mode
IF CR4.CET = 1 AND IA32_U_CET.SH_STK_EN = 1 AND newCS.L = 0 AND IA32_PL3_SSP[63:32] ≠ 0
    THEN #GP(0);
FI;
// If supervisor shadow stacks are enabled, compare SSP to the FRED SSP MSR for stack level 0
IF CR4.CET = 1 AND IA32_S_CET.SH_STK_EN = 1 AND IA32_FRED_SSP0 ≠ SSP
    THEN #CP(FAR-RET/IRET);
FI;
// update registers for return state
RIP := newRIP;
RFLAGS := newRFLAGS;                    // ERETU can set RFLAGS.RF to 1
RSP := newRSP;                          // load all 64 bits regardless of new mode
CS := newCS;                            // selector and descriptor
SS := newSS;                            // selector and descriptor
CPL := 3;
// swap GS.base and IA32_KERNEL_GS_BASE
tempGSB := GS.base;
GS.base := IA32_KERNEL_GS_BASE;
IA32_KERNEL_GS_BASE := tempGSB;
IF CR4.CET = 1 AND IA32_U_CET.SH_STK_EN = 1
    THEN SSP := IA32_PL3_SSP;
FI;
// update event-related state
IF NMI_unblock = 1
    THEN unblock NMIs;
FI;
IF pend_DB = 1 AND RFLAGS.TF =1
    THEN pend a single-step debug exception (#DB) to be delivered after ERETU;
FI;
```

Flags Affected

All defined flags and fields in the RFLAGS register are potentially modified except for the VM flag.

Protected Mode Exceptions

#UD	The ERETU instruction is not recognized in protected mode.

Real-Address Mode Exceptions

#UD	The ERETU instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The ERETU instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	The ERETU instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD	If CR4.FRED = 0.
	If CPL > 0.
	If the LOCK prefix is used.
#GP(0)	If CSL > 0.
	If the return is to 64-bit mode and the RIP image on the stack is not paging canonical.
	If the return is to compatibility mode and the low 32 bits of the RIP image on the stack is not within the CS.limit being established.
	If the RFLAGS image on the stack sets reserved bits or the VM bit (or fails to set bit 1).
	If bits 13:12 of the RFLAGS image on the stack (corresponding to IOPL) are not zero.
	If the value of bits 1:0 of the CS image on the stack (the target CPL) is not 3.
	If the value of bits 63:16 of the CS image on the stack is not zero.
	If the value of bits 1:0 of the SS image on the stack (the target CPL) is not 3.
	If the SS image on the stack sets undefined bits (bits 31:19).
	If the return is to compatibility mode and would load SSP with a value that sets any of bits 63:32.
	If ERETU is loading CS and SS from descriptor tables (see Operation section) and either of the following apply: • If the value of bits 15:2 of the CS image on the stack is 0. • If the value of bits 15:2 of the SS image on the stack is 0.
#GP(selector)	If ERETU is loading CS and SS from descriptor tables (see Operation section) and any of the following apply: • A segment selector index is outside its descriptor table limits. • A segment descriptor memory address is not paging canonical. • The segment descriptor for CS does not indicate that it is a code segment. • The segment descriptor for CS has both the D-bit and L-bit set. • The segment descriptor for CS is non-conforming and its DPL is not 3. • The segment descriptor for SS does not indicate that it is a writable data segment. • The DPL of the segment descriptor for SS is not 3.
#SS(0)	If an ordinary stack access would use an address that is not paging canonical or would cause a LASS violation.
#CP(FAR-RET/IRET)	If the value in SSP differs from that in IA32_FRED_SSP0.

EXTRACTPS—Extract Packed Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 17 /r ib EXTRACTPS reg/m32, xmm1, imm8	A	V/V	SSE4_1	Extract one single precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.
VEX.128.66.0F3A.WIG 17 /r ib VEXTRACTPS reg/m32, xmm1, imm8	A	V/V	AVX	Extract one single precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.
EVEX.128.66.0F3A.WIG 17 /r ib VEXTRACTPS reg/m32, xmm1, imm8	B	V/V	AVX512F OR AVX10.1	Extract one single precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
B	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A

Description

Extracts a single precision floating-point value from the source operand (second operand) at the 32-bit offset specified from imm8. Immediate bits higher than the most significant offset for the vector length are ignored.

The extracted single precision floating-point value is stored in the low 32-bits of the destination operand

In 64-bit mode, destination register operand has default operand size of 64 bits. The upper 32-bits of the register are filled with zero. REX.W is ignored.

VEX.128 and EVEX encoded version: When VEX.W1 or EVEX.W1 form is used in 64-bit mode with a general purpose register (GPR) as a destination operand, the packed single quantity is zero extended to 64 bits.

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

128-bit Legacy SSE version: When a REX.W prefix is used in 64-bit mode with a general purpose register (GPR) as a destination operand, the packed single quantity is zero extended to 64 bits.

The source register is an XMM register. Imm8[1:0] determine the starting DWORD offset from which to extract the 32-bit floating-point value.

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

Operation

VEXTRACTPS (EVEX and VEX.128 Encoded Version)

```
SRC_OFFSET := IMM8[1:0]
IF (64-Bit Mode and DEST is register)
    DEST[31:0] := (SRC[127:0] >> (SRC_OFFSET*32)) AND 0FFFFFFFFh
    DEST[63:32] := 0
ELSE
    DEST[31:0] := (SRC[127:0] >> (SRC_OFFSET*32)) AND 0FFFFFFFFh
FI
```

EXTRACTPS (128-bit Legacy SSE Version)

```
SRC_OFFSET := IMM8[1:0]
IF (64-Bit Mode and DEST is register)
    DEST[31:0] := (SRC[127:0] >> (SRC_OFFSET*32)) AND 0FFFFFFFFh
    DEST[63:32] := 0
ELSE
    DEST[31:0] := (SRC[127:0] >> (SRC_OFFSET*32)) AND 0FFFFFFFFh
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

EXTRACTPS int _mm_extract_ps (__m128 a, const int nidx);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instructions, see Table 2-22, "Type 5 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-59, "Type E9NF Class Exception Conditions."

Additionally:

#UD	IF VEX.L = 0.
#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

F2XM1—Compute 2^x-1

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F0	F2XM1	Valid	Valid	Replace ST(0) with $(2^{ST(0)} - 1)$.

Description

Computes the exponential value of 2 to the power of the source operand minus 1. The source operand is located in register ST(0) and the result is also stored in ST(0). The value of the source operand must lie in the range –1.0 to +1.0. If the source value is outside this range, the result is undefined.

The following table shows the results obtained when computing the exponential value of various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-18. Results Obtained from F2XM1

ST(0) SRC	ST(0) DEST
– 1.0 to –0	– 0.5 to – 0
– 0	– 0
+ 0	+ 0
+ 0 to +1.0	+ 0 to 1.0

Values other than 2 can be exponentiated using the following formula:

$$x^y := 2^{(y * \log_2 x)}$$

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

Operation

$ST(0) := (2^{ST(0)} - 1);$

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Source is a denormal value.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FABS—Absolute Value

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 E1	FABS	Valid	Valid	Replace ST with its absolute value.

Description

Clears the sign bit of ST(0) to create the absolute value of the operand. The following table shows the results obtained when creating the absolute value of various classes of numbers.

Table 3-19. Results Obtained from FABS

ST(0) SRC	ST(0) DEST
−∞	+∞
−F	+F
−0	+0
+0	+0
+F	+F
+∞	+∞
NaN	NaN

NOTES:

F Means finite floating-point value.

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

Operation

```
ST(0) := |ST(0)|;
```

FPU Flags Affected

C1	Set to 0.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FADD/FADDP/FIADD—Add

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D8 /0	FADD m32fp	Valid	Valid	Add m32fp to ST(0) and store result in ST(0).
DC /0	FADD m64fp	Valid	Valid	Add m64fp to ST(0) and store result in ST(0).
D8 C0+i	FADD ST(0), ST(i)	Valid	Valid	Add ST(0) to ST(i) and store result in ST(0).
DC C0+i	FADD ST(i), ST(0)	Valid	Valid	Add ST(i) to ST(0) and store result in ST(i).
DE C0+i	FADDP ST(i), ST(0)	Valid	Valid	Add ST(0) to ST(i), store result in ST(i), and pop the register stack.
DE C1	FADDP	Valid	Valid	Add ST(0) to ST(1), store result in ST(1), and pop the register stack.
DA /0	FIADD m32int	Valid	Valid	Add m32int to ST(0) and store result in ST(0).
DE /0	FIADD m16int	Valid	Valid	Add m16int to ST(0) and store result in ST(0).

Description

Adds the destination and source operands and stores the sum in the destination location. The destination operand is always an FPU register; the source operand can be a register or a memory location. Source operands in memory can be in single precision or double precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction adds the contents of the ST(0) register to the ST(1) register. The one-operand version adds the contents of a memory location (either a floating-point or an integer value) to the contents of the ST(0) register. The two-operand version, adds the contents of the ST(0) register to the ST(i) register or vice versa. The value in ST(0) can be doubled by coding:

```
FADD ST(0), ST(0);
```

The FADDP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. (The no-operand version of the floating-point add instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FADD rather than FADDP.)

The FIADD instructions convert an integer source operand to double extended-precision floating-point format before performing the addition.

The table on the following page shows the results obtained when adding various classes of numbers, assuming that neither overflow nor underflow occurs.

When the sum of two operands with opposite signs is 0, the result is +0, except for the round toward $-\infty$ mode, in which case the result is –0. When the source operand is an integer 0, it is treated as a +0.

When both operand are infinities of the same sign, the result is ∞ of the expected sign. If both operands are infinities of opposite signs, an invalid-operation exception is generated. See Table 3-20.

Table 3-20. FADD/FADDP/FIADD Results

		DEST						
		−∞	−F	−0	+0	+F	+∞	NaN
SRC	−∞	−∞	−∞	−∞	−∞	−∞	*	NaN
	−F or −I	−∞	−F	SRC	SRC	±F or ±0	+∞	NaN
	−0	−∞	DEST	−0	±0	DEST	+∞	NaN
	+0	−∞	DEST	±0	+0	DEST	+∞	NaN
	+F or +I	−∞	±F or ±0	SRC	SRC	+F	+∞	NaN
	+∞	*	+∞	+∞	+∞	+∞	+∞	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

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

Operation

```
IF Instruction = FIADD
    THEN
        DEST := DEST + ConvertToDoubleExtendedPrecisionFP(SRC);
    ELSE (* Source operand is floating-point value *)
        DEST := DEST + SRC;
FI;
IF Instruction = FADDP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format.
	Operands are infinities of unlike sign.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FBLD—Load Binary Coded Decimal

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DF /4	FBLD m80bcd	Valid	Valid	Convert BCD value to floating-point and push onto the FPU stack.

Description

Converts the BCD source operand into double extended-precision floating-point format and pushes the value onto the FPU stack. The source operand is loaded without rounding errors. The sign of the source operand is preserved, including that of –0.

The packed BCD digits are assumed to be in the range 0 through 9; the instruction does not check for invalid digits (AH through FH). Attempting to load an invalid encoding produces an undefined result.

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

Operation

```
TOP := TOP – 1;
ST(0) := ConvertToDoubleExtendedPrecisionFP(SRC);
```

FPU Flags Affected

C1	Set to 1 if stack overflow occurred; otherwise, set to 0.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack overflow occurred.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FBSTP—Store BCD Integer and Pop

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DF /6	FBSTP m80bcd	Valid	Valid	Store ST(0) in m80bcd and pop ST(0).

Description

Converts the value in the ST(0) register to an 18-digit packed BCD integer, stores the result in the destination operand, and pops the register stack. If the source value is a non-integral value, it is rounded to an integer value, according to rounding mode specified by the RC field of the FPU control word. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

The destination operand specifies the address where the first byte destination value is to be stored. The BCD value (including its sign bit) requires 10 bytes of space in memory.

The following table shows the results obtained when storing various classes of numbers in packed BCD format.

Table 3-21. FBSTP Results

ST(0)	DEST
– ∞ or Value Too Large for DEST Format	*
F ≤ – 1	– D
–1 < F < -0	**
– 0	– 0
+ 0	+ 0
+ 0 < F < +1	**
F ≥ +1	+ D
+ ∞ or Value Too Large for DEST Format	*
NaN	*

NOTES:

F Means finite floating-point value.

D Means packed-BCD number.

* Indicates floating-point invalid-operation (#IA) exception.

** ±0 or ±1, depending on the rounding mode.

If the converted value is too large for the destination format, or if the source operand is an ∞, SNaN, QNAN, or is in an unsupported format, an invalid-arithmetic-operand condition is signaled. If the invalid-operation exception is not masked, an invalid-arithmetic-operand exception (#IA) is generated and no value is stored in the destination operand. If the invalid-operation exception is masked, the packed BCD indefinite value is stored in memory.

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

Operation

```
DEST := BCD(ST(0));
PopRegisterStack;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Converted value that exceeds 18 BCD digits in length.
	Source operand is an SNaN, QNaN, ±∞, or in an unsupported format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If a segment register is being loaded with a segment selector that points to a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FCHS—Change Sign

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 E0	FCHS	Valid	Valid	Complements sign of ST(0).

Description

Complements the sign bit of ST(0). This operation changes a positive value into a negative value of equal magnitude or vice versa. The following table shows the results obtained when changing the sign of various classes of numbers.

Table 3-22. FCHS Results

ST(0) SRC	ST(0) DEST
– ∞	+ ∞
– F	+ F
– 0	+ 0
+ 0	– 0
+ F	– F
+ ∞	– ∞
NaN	NaN

NOTES:

* F means finite floating-point value.

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

Operation

```
SignBit(ST(0)) := NOT (SignBit(ST(0)));
```

FPU Flags Affected

C1	Set to 0.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FCLEX/FNCLEX—Clear Exceptions

Opcode[1]	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
9B DB E2	FCLEX	Valid	Valid	Clear floating-point exception flags after checking for pending unmasked floating-point exceptions.
DB E2	FNCLEX[1]	Valid	Valid	Clear floating-point exception flags without checking for pending unmasked floating-point exceptions.

NOTES:

1. See IA-32 Architecture Compatibility section below.

Description

Clears the floating-point exception flags (PE, UE, OE, ZE, DE, and IE), the exception summary status flag (ES), the stack fault flag (SF), and the busy flag (B) in the FPU status word. The FCLEX instruction checks for and handles any pending unmasked floating-point exceptions before clearing the exception flags; the FNCLEX instruction does not.

The assembler issues two instructions for the FCLEX instruction (an FWAIT instruction followed by an FNCLEX instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS* compatibility mode, it is possible (under unusual circumstances) for an FNCLEX instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for a description of these circumstances. An FNCLEX instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

This instruction affects only the x87 FPU floating-point exception flags. It does not affect the SIMD floating-point exception flags in the MXCSR register.

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

Operation

```
FPUStatusWord[0:7] := 0;
FPUStatusWord[15] := 0;
```

FPU Flags Affected

The PE, UE, OE, ZE, DE, IE, ES, SF, and B flags in the FPU status word are cleared. The C0, C1, C2, and C3 flags are undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FCMOV*cc*—Floating-Point Conditional Move

Opcode[1]	Instruction	64-Bit Mode	Compat/ Leg Mode[1]	Description
DA C0+i	FCMOVB ST(0), ST(i)	Valid	Valid	Move if below (CF=1).
DA C8+i	FCMOVE ST(0), ST(i)	Valid	Valid	Move if equal (ZF=1).
DA D0+i	FCMOVBE ST(0), ST(i)	Valid	Valid	Move if below or equal (CF=1 or ZF=1).
DA D8+i	FCMOVU ST(0), ST(i)	Valid	Valid	Move if unordered (PF=1).
DB C0+i	FCMOVNB ST(0), ST(i)	Valid	Valid	Move if not below (CF=0).
DB C8+i	FCMOVNE ST(0), ST(i)	Valid	Valid	Move if not equal (ZF=0).
DB D0+i	FCMOVNBE ST(0), ST(i)	Valid	Valid	Move if not below or equal (CF=0 and ZF=0).
DB D8+i	FCMOVNU ST(0), ST(i)	Valid	Valid	Move if not unordered (PF=0).

NOTES:

1. See IA-32 Architecture Compatibility section below.

Description

Tests the status flags in the EFLAGS register and moves the source operand (second operand) to the destination operand (first operand) if the given test condition is true. The condition for each mnemonic os given in the Description column above and in Chapter 8 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1. The source operand is always in the ST(i) register and the destination operand is always ST(0).

The FCMOV*cc* instructions are useful for optimizing small IF constructions. They also help eliminate branching overhead for IF operations and the possibility of branch mispredictions by the processor.

A processor may not support the FCMOV*cc* instructions. Software can check if the FCMOV*cc* instructions are supported by checking the processor's feature information with the CPUID instruction (see "COMISS—Compare Scalar Ordered Single Precision Floating-Point Values and Set EFLAGS" in this chapter). If both the CMOV and FPU feature bits are set, the FCMOV*cc* instructions are supported.

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

IA-32 Architecture Compatibility

The FCMOVcc instructions were introduced to the IA-32 Architecture in the P6 family processors and are not available in earlier IA-32 processors.

Operation

```
IF condition TRUE
    THEN ST(0) := ST(i);
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.

Integer Flags Affected

None.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FCOM/FCOMP/FCOMPP—Compare Floating-Point Values

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D8 /2	FCOM m32fp	Valid	Valid	Compare ST(0) with m32fp.
DC /2	FCOM m64fp	Valid	Valid	Compare ST(0) with m64fp.
D8 D0+i	FCOM ST(i)	Valid	Valid	Compare ST(0) with ST(i).
D8 D1	FCOM	Valid	Valid	Compare ST(0) with ST(1).
D8 /3	FCOMP m32fp	Valid	Valid	Compare ST(0) with m32fp and pop register stack.
DC /3	FCOMP m64fp	Valid	Valid	Compare ST(0) with m64fp and pop register stack.
D8 D8+i	FCOMP ST(i)	Valid	Valid	Compare ST(0) with ST(i) and pop register stack.
D8 D9	FCOMP	Valid	Valid	Compare ST(0) with ST(1) and pop register stack.
DE D9	FCOMPP	Valid	Valid	Compare ST(0) with ST(1) and pop register stack twice.

Description

Compares the contents of register ST(0) and source value and sets condition code flags C0, C2, and C3 in the FPU status word according to the results (see the table below). The source operand can be a data register or a memory location. If no source operand is given, the value in ST(0) is compared with the value in ST(1). The sign of zero is ignored, so that −0.0 is equal to +0.0.

Table 3-23. FCOM/FCOMP/FCOMPP Results

Condition	C3	C2	C0
ST(0) > SRC	0	0	0
ST(0) < SRC	0	0	1
ST(0) = SRC	1	0	0
Unordered*	1	1	1

NOTES:

* Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

This instruction checks the class of the numbers being compared (see "FXAM—Examine Floating-Point" in this chapter). If either operand is a NaN or is in an unsupported format, an invalid-arithmetic-operand exception (#IA) is raised and, if the exception is masked, the condition flags are set to "unordered." If the invalid-arithmetic-operand exception is unmasked, the condition code flags are not set.

The FCOMP instruction pops the register stack following the comparison operation and the FCOMPP instruction pops the register stack twice following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

The FCOM instructions perform the same operation as the FUCOM instructions. The only difference is how they handle QNaN operands. The FCOM instructions raise an invalid-arithmetic-operand exception (#IA) when either or both of the operands is a NaN value or is in an unsupported format. The FUCOM instructions perform the same operation as the FCOM instructions, except that they do not generate an invalid-arithmetic-operand exception for QNaNs.

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

Operation

```
CASE (relation of operands) OF
    ST > SRC:       C3, C2, C0 := 000;
    ST < SRC:       C3, C2, C0 := 001;
    ST = SRC:       C3, C2, C0 := 100;
ESAC;

IF ST(0) or SRC = NaN or unsupported format
    THEN
        #IA
        IF FPUControlWord.IM = 1
            THEN
                C3, C2, C0 := 111;
        FI;
FI;

IF Instruction = FCOMP
    THEN
        PopRegisterStack;
FI;

IF Instruction = FCOMPP
    THEN
        PopRegisterStack;
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0.
C0, C2, C3	See table on previous page.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	One or both operands are NaN values or have unsupported formats. Register is marked empty.
#D	One or both operands are denormal values.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FCOMI/FCOMIP/FUCOMI/FUCOMIP—Compare Floating-Point Values and Set EFLAGS

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DB F0+i	FCOMI ST, ST(i)	Valid	Valid	Compare ST(0) with ST(i) and set status flags accordingly.
DF F0+i	FCOMIP ST, ST(i)	Valid	Valid	Compare ST(0) with ST(i), set status flags accordingly, and pop register stack.
DB E8+i	FUCOMI ST, ST(i)	Valid	Valid	Compare ST(0) with ST(i), check for ordered values, and set status flags accordingly.
DF E8+i	FUCOMIP ST, ST(i)	Valid	Valid	Compare ST(0) with ST(i), check for ordered values, set status flags accordingly, and pop register stack.

Description

Performs an unordered comparison of the contents of registers ST(0) and ST(i) and sets the status flags ZF, PF, and CF in the EFLAGS register according to the results (see the table below). The sign of zero is ignored for comparisons, so that −0.0 is equal to +0.0.

Table 3-24. FCOMI/FCOMIP/ FUCOMI/FUCOMIP Results

Comparison Results*	ZF	PF	CF
ST0 > ST(i)	0	0	0
ST0 < ST(i)	0	0	1
ST0 = ST(i)	1	0	0
Unordered**	1	1	1

NOTES:

* See the IA-32 Architecture Compatibility section below.

** Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

An unordered comparison checks the class of the numbers being compared (see "FXAM—Examine Floating-Point" in this chapter). The FUCOMI/FUCOMIP instructions perform the same operations as the FCOMI/FCOMIP instructions. The only difference is that the FUCOMI/FUCOMIP instructions raise the invalid-arithmetic-operand exception (#IA) only when either or both operands are an SNaN or are in an unsupported format; QNaNs cause the condition code flags to be set to unordered, but do not cause an exception to be generated. The FCOMI/FCOMIP instructions raise an invalid-operation exception when either or both of the operands are a NaN value of any kind or are in an unsupported format.

If the operation results in an invalid-arithmetic-operand exception being raised, the status flags in the EFLAGS register are set only if the exception is masked.

The FCOMI/FCOMIP and FUCOMI/FUCOMIP instructions set the OF, SF, and AF flags to zero in the EFLAGS register (regardless of whether an invalid-operation exception is detected).

The FCOMIP and FUCOMIP instructions also pop the register stack following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

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

IA-32 Architecture Compatibility

The FCOMI/FCOMIP/FUCOMI/FUCOMIP instructions were introduced to the IA-32 Architecture in the P6 family processors and are not available in earlier IA-32 processors.

Operation

```
CASE (relation of operands) OF
    ST(0) > ST(i):      ZF, PF, CF := 000;
    ST(0) < ST(i):      ZF, PF, CF := 001;
    ST(0) = ST(i):      ZF, PF, CF := 100;
ESAC;

IF Instruction is FCOMI or FCOMIP
    THEN
        IF ST(0) or ST(i) = NaN or unsupported format
            THEN
                #IA
                IF FPUControlWord.IM = 1
                    THEN
                        ZF, PF, CF := 111;
                FI;
        FI;
FI;

IF Instruction is FUCOMI or FUCOMIP
    THEN
        IF ST(0) or ST(i) = QNaN, but not SNaN or unsupported format
            THEN
                ZF, PF, CF := 111;
            ELSE (* ST(0) or ST(i) is SNaN or unsupported format *)
                 #IA;
                IF FPUControlWord.IM = 1
                    THEN
                        ZF, PF, CF := 111;
                FI;
        FI;
FI;

IF Instruction is FCOMIP or FUCOMIP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0.
C0, C2, C3	Not affected.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	(FCOMI or FCOMIP instruction) One or both operands are NaN values or have unsupported formats. (FUCOMI or FUCOMIP instruction) One or both operands are SNaN values (but not QNaNs) or have undefined formats. Detection of a QNaN value does not raise an invalid-operand exception.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FCOS—Cosine

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 FF	FCOS	Valid	Valid	Replace ST(0) with its approximate cosine.

Description

Computes the approximate cosine of the source operand in register ST(0) and stores the result in ST(0). The source operand must be given in radians and must be within the range -2^{63} to $+2^{63}$. The following table shows the results obtained when taking the cosine of various classes of numbers.

Table 3-25. FCOS Results

ST(0) SRC	ST(0) DEST
$-\infty$	*
−F	−1 to +1
−0	+1
+0	+1
+F	−1 to +1
$+\infty$	*
NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range -2^{63} to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of 2π. However, even within the range -2^{63} to $+2^{63}$, inaccurate results can occur because the finite approximation of π used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FCOS only to arguments reduced accurately in software, to a value smaller in absolute value than $3\pi/8$. See the sections titled "Approximation of Pi" and "Transcendental Instruction Accuracy" in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for a discussion of the proper value to use for π in performing such reductions.

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

Operation

```
IF |ST(0)| < 2^63
THEN
    C2 := 0;
    ST(0) := FCOS(ST(0)); // approximation of cosine
ELSE (* Source operand is out-of-range *)
    C2 := 1;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise. Undefined if C2 is 1.
C2	Set to 1 if outside range (-2^{63} < source operand < $+2^{63}$); otherwise, set to 0.
C0, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, ∞, or unsupported format.
#D	Source is a denormal value.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FDECSTP—Decrement Stack-Top Pointer

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F6	FDECSTP	Valid	Valid	Decrement TOP field in FPU status word.

Description

Subtracts one from the TOP field of the FPU status word (decrements the top-of-stack pointer). If the TOP field contains a 0, it is set to 7. The effect of this instruction is to rotate the stack by one position. The contents of the FPU data registers and tag register are not affected.

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

Operation

```
IF TOP = 0
    THEN TOP := 7;
    ELSE TOP := TOP - 1;
FI;
```

FPU Flags Affected

The C1 flag is set to 0. The C0, C2, and C3 flags are undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FDIV/FDIVP/FIDIV—Divide

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D8 /6	FDIV m32fp	Valid	Valid	Divide ST(0) by m32fp and store result in ST(0).
DC /6	FDIV m64fp	Valid	Valid	Divide ST(0) by m64fp and store result in ST(0).
D8 F0+i	FDIV ST(0), ST(i)	Valid	Valid	Divide ST(0) by ST(i) and store result in ST(0).
DC F8+i	FDIV ST(i), ST(0)	Valid	Valid	Divide ST(i) by ST(0) and store result in ST(i).
DE F8+i	FDIVP ST(i), ST(0)	Valid	Valid	Divide ST(i) by ST(0), store result in ST(i), and pop the register stack.
DE F9	FDIVP	Valid	Valid	Divide ST(1) by ST(0), store result in ST(1), and pop the register stack.
DA /6	FIDIV m32int	Valid	Valid	Divide ST(0) by m32int and store result in ST(0).
DE /6	FIDIV m16int	Valid	Valid	Divide ST(0) by m16int and store result in ST(0).

Description

Divides the destination operand by the source operand and stores the result in the destination location. The destination operand (dividend) is always in an FPU register; the source operand (divisor) can be a register or a memory location. Source operands in memory can be in single precision or double precision floating-point format, word or doubleword integer format.

The no-operand version of the instruction divides the contents of the ST(1) register by the contents of the ST(0) register. The one-operand version divides the contents of the ST(0) register by the contents of a memory location (either a floating-point or an integer value). The two-operand version, divides the contents of the ST(0) register by the contents of the ST(i) register or vice versa.

The FDIVP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point divide instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FDIV rather than FDIVP.

The FIDIV instructions convert an integer source operand to double extended-precision floating-point format before performing the division. When the source operand is an integer 0, it is treated as a +0.

If an unmasked divide-by-zero exception (#Z) is generated, no result is stored; if the exception is masked, an ∞ of the appropriate sign is stored in the destination operand.

The following table shows the results obtained when dividing various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-26. FDIV/FDIVP/FIDIV Results

		DEST						
		− ∞	− F	− 0	+ 0	+ F	+ ∞	NaN
SRC	− ∞	*	+ 0	+ 0	− 0	− 0	*	NaN
	− F	+ ∞	+ F	+ 0	− 0	− F	− ∞	NaN
	− I	+ ∞	+ F	+ 0	− 0	− F	− ∞	NaN
	− 0	+ ∞	**	*	*	**	− ∞	NaN
	+ 0	− ∞	**	*	*	**	+ ∞	NaN
	+ I	− ∞	− F	− 0	+ 0	+ F	+ ∞	NaN
	+ F	− ∞	− F	− 0	+ 0	+ F	+ ∞	NaN
	+ ∞	*	− 0	− 0	+ 0	+ 0	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

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

Operation

```
IF SRC = 0
    THEN
        #Z;
    ELSE
        IF Instruction is FIDIV
            THEN
                DEST := DEST / ConvertToDoubleExtendedPrecisionFP(SRC);
            ELSE (* Source operand is floating-point value *)
                DEST := DEST / SRC;
        FI;
FI;
IF Instruction = FDIVP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format. ±∞ / ±∞; ±0 / ±0
#D	Source is a denormal value.
#Z	DEST / ±0, where DEST is not equal to ±0.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FDIVR/FDIVRP/FIDIVR—Reverse Divide

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D8 /7	FDIVR m32fp	Valid	Valid	Divide m32fp by ST(0) and store result in ST(0).
DC /7	FDIVR m64fp	Valid	Valid	Divide m64fp by ST(0) and store result in ST(0).
D8 F8+i	FDIVR ST(0), ST(i)	Valid	Valid	Divide ST(i) by ST(0) and store result in ST(0).
DC F0+i	FDIVR ST(i), ST(0)	Valid	Valid	Divide ST(0) by ST(i) and store result in ST(i).
DE F0+i	FDIVRP ST(i), ST(0)	Valid	Valid	Divide ST(0) by ST(i), store result in ST(i), and pop the register stack.
DE F1	FDIVRP	Valid	Valid	Divide ST(0) by ST(1), store result in ST(1), and pop the register stack.
DA /7	FIDIVR m32int	Valid	Valid	Divide m32int by ST(0) and store result in ST(0).
DE /7	FIDIVR m16int	Valid	Valid	Divide m16int by ST(0) and store result in ST(0).

Description

Divides the source operand by the destination operand and stores the result in the destination location. The destination operand (divisor) is always in an FPU register; the source operand (dividend) can be a register or a memory location. Source operands in memory can be in single precision or double precision floating-point format, word or doubleword integer format.

These instructions perform the reverse operations of the FDIV, FDIVP, and FIDIV instructions. They are provided to support more efficient coding.

The no-operand version of the instruction divides the contents of the ST(0) register by the contents of the ST(1) register. The one-operand version divides the contents of a memory location (either a floating-point or an integer value) by the contents of the ST(0) register. The two-operand version, divides the contents of the ST(i) register by the contents of the ST(0) register or vice versa.

The FDIVRP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point divide instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FDIVR rather than FDIVRP.

The FIDIVR instructions convert an integer source operand to double extended-precision floating-point format before performing the division.

If an unmasked divide-by-zero exception (#Z) is generated, no result is stored; if the exception is masked, an ∞ of the appropriate sign is stored in the destination operand.

The following table shows the results obtained when dividing various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-27. FDIVR/FDIVRP/FIDIVR Results

		DEST						
		− ∞	− F	− 0	+ 0	+ F	+ ∞	NaN
SRC	− ∞	*	+ ∞	+ ∞	− ∞	− ∞	*	NaN
	− F	+ 0	+ F	**	**	− F	− 0	NaN
	− I	+ 0	+ F	**	**	− F	− 0	NaN
	− 0	+ 0	+ 0	*	*	− 0	− 0	NaN
	+ 0	− 0	− 0	*	*	+ 0	+ 0	NaN
	+ I	− 0	− F	**	**	+ F	+ 0	NaN
	+ F	− 0	− F	**	**	+ F	+ 0	NaN
	+ ∞	*	− ∞	− ∞	+ ∞	+ ∞	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

When the source operand is an integer 0, it is treated as a +0. This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
IF DEST = 0
    THEN
        #Z;
    ELSE
        IF Instruction = FIDIVR
            THEN
                DEST := ConvertToDoubleExtendedPrecisionFP(SRC) / DEST;
            ELSE (* Source operand is floating-point value *)
                DEST := SRC / DEST;
        FI;
FI;
IF Instruction = FDIVRP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format.
	±∞ / ±∞; ±0 / ±0
#D	Source is a denormal value.
#Z	SRC / ±0, where SRC is not equal to ±0.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FFREE—Free Floating-Point Register

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DD C0+i	FFREE ST(i)	Valid	Valid	Sets tag for ST(i) to empty.

Description

Sets the tag in the FPU tag register associated with register ST(i) to empty (11B). The contents of ST(i) and the FPU stack-top pointer (TOP) are not affected.

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

Operation

```
TAG(i) := 11B;
```

FPU Flags Affected

C0, C1, C2, C3 undefined.

Floating-Point Exceptions

None

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FICOM/FICOMP—Compare Integer

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DE /2	FICOM m16int	Valid	Valid	Compare ST(0) with m16int.
DA /2	FICOM m32int	Valid	Valid	Compare ST(0) with m32int.
DE /3	FICOMP m16int	Valid	Valid	Compare ST(0) with m16int and pop stack register.
DA /3	FICOMP m32int	Valid	Valid	Compare ST(0) with m32int and pop stack register.

Description

Compares the value in ST(0) with an integer source operand and sets the condition code flags C0, C2, and C3 in the FPU status word according to the results (see table below). The integer value is converted to double extended-precision floating-point format before the comparison is made.

Table 3-28. FICOM/FICOMP Results

Condition	C3	C2	C0
ST(0) > SRC	0	0	0
ST(0) < SRC	0	0	1
ST(0) = SRC	1	0	0
Unordered	1	1	1

These instructions perform an "unordered comparison." An unordered comparison also checks the class of the numbers being compared (see "FXAM—Examine Floating-Point" in this chapter). If either operand is a NaN or is in an undefined format, the condition flags are set to "unordered."

The sign of zero is ignored, so that −0.0 := +0.0.

The FICOMP instructions pop the register stack following the comparison. To pop the register stack, the processor marks the ST(0) register empty and increments the stack pointer (TOP) by 1.

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

Operation

```
CASE (relation of operands) OF
    ST(0) > SRC:    C3, C2, C0 := 000;
    ST(0) < SRC:    C3, C2, C0 := 001;
    ST(0) = SRC:    C3, C2, C0 := 100;
    Unordered:      C3, C2, C0 := 111;
ESAC;

IF Instruction = FICOMP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0.
C0, C2, C3	See table on previous page.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	One or both operands are NaN values or have unsupported formats.
#D	One or both operands are denormal values.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FILD—Load Integer

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DF /0	FILD m16int	Valid	Valid	Push m16int onto the FPU register stack.
DB /0	FILD m32int	Valid	Valid	Push m32int onto the FPU register stack.
DF /5	FILD m64int	Valid	Valid	Push m64int onto the FPU register stack.

Description

Converts the signed-integer source operand into double extended-precision floating-point format and pushes the value onto the FPU register stack. The source operand can be a word, doubleword, or quadword integer. It is loaded without rounding errors. The sign of the source operand is preserved.

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

Operation

```
TOP := TOP – 1;
ST(0) := ConvertToDoubleExtendedPrecisionFP(SRC);
```

FPU Flags Affected

C1	Set to 1 if stack overflow occurred; set to 0 otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack overflow occurred.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FINCSTP—Increment Stack-Top Pointer

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F7	FINCSTP	Valid	Valid	Increment the TOP field in the FPU status register.

Description

Adds one to the TOP field of the FPU status word (increments the top-of-stack pointer). If the TOP field contains a 7, it is set to 0. The effect of this instruction is to rotate the stack by one position. The contents of the FPU data registers and tag register are not affected. This operation is not equivalent to popping the stack, because the tag for the previous top-of-stack register is not marked empty.

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

Operation

```
IF TOP = 7
    THEN TOP := 0;
    ELSE TOP := TOP + 1;
FI;
```

FPU Flags Affected

The C1 flag is set to 0. The C0, C2, and C3 flags are undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FINIT/FNINIT—Initialize Floating-Point Unit

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
9B DB E3	FINIT	Valid	Valid	Initialize FPU after checking for pending unmasked floating-point exceptions.
DB E3	FNINIT[1]	Valid	Valid	Initialize FPU without checking for pending unmasked floating-point exceptions.

NOTES:

1. See IA-32 Architecture Compatibility section below.

Description

Sets the FPU control, status, tag, instruction pointer, and data pointer registers to their default states. The FPU control word is set to 037FH (round to nearest, all exceptions masked, 64-bit precision). The status word is cleared (no exception flags set, TOP is set to 0). The data registers in the register stack are left unchanged, but they are all tagged as empty (11B). Both the instruction and data pointers are cleared.

The FINIT instruction checks for and handles any pending unmasked floating-point exceptions before performing the initialization; the FNINIT instruction does not.

The assembler issues two instructions for the FINIT instruction (an FWAIT instruction followed by an FNINIT instruction), and the processor executes each of these instructions in separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

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

IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNINIT instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for a description of these circumstances. An FNINIT instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

In the Intel387 math coprocessor, the FINIT/FNINIT instruction does not clear the instruction and data pointers.

This instruction affects only the x87 FPU. It does not affect the XMM and MXCSR registers.

Operation

```
FPUControlWord := 037FH;
FPUStatusWord := 0;
FPUTagWord := FFFFH;
FPUDataPointer := 0;
FPUInstructionPointer := 0;
FPULastInstructionOpcode := 0;
```

FPU Flags Affected

C0, C1, C2, C3 set to 0.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FIST/FISTP—Store Integer

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DF /2	FIST m16int	Valid	Valid	Store ST(0) in m16int.
DB /2	FIST m32int	Valid	Valid	Store ST(0) in m32int.
DF /3	FISTP m16int	Valid	Valid	Store ST(0) in m16int and pop register stack.
DB /3	FISTP m32int	Valid	Valid	Store ST(0) in m32int and pop register stack.
DF /7	FISTP m64int	Valid	Valid	Store ST(0) in m64int and pop register stack.

Description

The FIST instruction converts the value in the ST(0) register to a signed integer and stores the result in the destination operand. Values can be stored in word or doubleword integer format. The destination operand specifies the address where the first byte of the destination value is to be stored.

The FISTP instruction performs the same operation as the FIST instruction and then pops the register stack. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The FISTP instruction also stores values in quadword integer format.

The following table shows the results obtained when storing various classes of numbers in integer format.

Table 3-29. FIST/FISTP Results

ST(0)	DEST
$-\infty$ or Value Too Large for DEST Format	*
$F \leq -1$	$-I$
$-1 < F < -0$	**
-0	0
$+0$	0
$+0 < F < +1$	**
$F \geq +1$	$+I$
$+\infty$ or Value Too Large for DEST Format	*
NaN	*

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-operation (#IA) exception.

** 0 or ±1, depending on the rounding mode.

If the source value is a non-integral value, it is rounded to an integer value, according to the rounding mode specified by the RC field of the FPU control word.

If the converted value is too large for the destination format, or if the source operand is an ∞, SNaN, QNAN, or is in an unsupported format, an invalid-arithmetic-operand condition is signaled. If the invalid-operation exception is not masked, an invalid-arithmetic-operand exception (#IA) is generated and no value is stored in the destination operand. If the invalid-operation exception is masked, the integer indefinite value is stored in memory.

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

Operation

```
DEST := Integer(ST(0));

IF Instruction = FISTP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Indicates rounding direction of if the inexact exception (#P) is generated: 0 := not roundup; 1 := roundup. Set to 0 otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Converted value is too large for the destination format. Source operand is an SNaN, QNaN, ±∞, or unsupported format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FISTTP—Store Integer With Truncation

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DF /1	FISTTP m16int	Valid	Valid	Store ST(0) in m16int with truncation.
DB /1	FISTTP m32int	Valid	Valid	Store ST(0) in m32int with truncation.
DD /1	FISTTP m64int	Valid	Valid	Store ST(0) in m64int with truncation.

Description

FISTTP converts the value in ST into a signed integer using truncation (chop) as rounding mode, transfers the result to the destination, and pop ST. FISTTP accepts word, short integer, and long integer destinations.

The following table shows the results obtained when storing various classes of numbers in integer format.

Table 3-30. FISTTP Results

ST(0)	DEST
$-\infty$ or Value Too Large for DEST Format	*
F ≤ -1	$-$I
$-1 <$ F $< +1$	0
F Š $+1$	$+$I
$+\infty$ or Value Too Large for DEST Format	*
NaN	*

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-operation (#IA) exception.

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

Operation

```
DEST := ST;
pop ST;
```

Flags Affected

C1 is cleared; C0, C2, C3 undefined.

Numeric Exceptions

Invalid, Stack Invalid (stack underflow), Precision.

Protected Mode Exceptions

#GP(0)	If the destination is in a nonwritable segment.
	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#NM	If CR0.EM[bit 2] = 1.
	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.SSE3[0] = 0.
	If the LOCK prefix is used.

Real Address Mode Exceptions

GP(0)	If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
#NM	If CR0.EM[bit 2] = 1.
	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.SSE3[0] = 0.
	If the LOCK prefix is used.

Virtual 8086 Mode Exceptions

GP(0)	If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
#NM	If CR0.EM[bit 2] = 1.
	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.SSE3[0] = 0.
	If the LOCK prefix is used.
#PF(fault-code)	For a page fault.
#AC(0)	For unaligned memory reference if the current privilege is 3.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
	If the LOCK prefix is used.

FLD—Load Floating-Point Value

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 /0	FLD m32fp	Valid	Valid	Push m32fp onto the FPU register stack.
DD /0	FLD m64fp	Valid	Valid	Push m64fp onto the FPU register stack.
DB /5	FLD m80fp	Valid	Valid	Push m80fp onto the FPU register stack.
D9 C0+i	FLD ST(i)	Valid	Valid	Push ST(i) onto the FPU register stack.

Description

Pushes the source operand onto the FPU register stack. The source operand can be in single precision, double precision, or double extended-precision floating-point format. If the source operand is in single precision or double precision floating-point format, it is automatically converted to the double extended-precision floating-point format before being pushed on the stack.

The FLD instruction can also push the value in a selected FPU register [ST(i)] onto the stack. Here, pushing register ST(0) duplicates the stack top.

NOTE

When the FLD instruction loads a denormal value and the DM bit in the CW is not masked, an exception is flagged but the value is still pushed onto the x87 stack.

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

Operation

```
IF SRC is ST(i)
    THEN
        temp := ST(i);
FI;
TOP := TOP – 1;
IF SRC is memory-operand
    THEN
        ST(0) := ConvertToDoubleExtendedPrecisionFP(SRC);
    ELSE (* SRC is ST(i) *)
        ST(0) := temp;
FI;
```

FPU Flags Affected

C1	Set to 1 if stack overflow occurred; otherwise, set to 0.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow or overflow occurred.
#IA	Source operand is an SNaN. Does not occur if the source operand is in double extended-precision floating-point format (FLD m80fp or FLD ST(i)).
#D	Source operand is a denormal value. Does not occur if the source operand is in double extended-precision floating-point format.

Protected Mode Exceptions

#GP(0)	If destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FLD1/FLDL2T/FLDL2E/FLDPI/FLDLG2/FLDLN2/FLDZ—Load Constant

Opcode*	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 E8	FLD1	Valid	Valid	Push +1.0 onto the FPU register stack.
D9 E9	FLDL2T	Valid	Valid	Push $\log_2 10$ onto the FPU register stack.
D9 EA	FLDL2E	Valid	Valid	Push $\log_2 e$ onto the FPU register stack.
D9 EB	FLDPI	Valid	Valid	Push π onto the FPU register stack.
D9 EC	FLDLG2	Valid	Valid	Push $\log_{10} 2$ onto the FPU register stack.
D9 ED	FLDLN2	Valid	Valid	Push $\log_e 2$ onto the FPU register stack.
D9 EE	FLDZ	Valid	Valid	Push +0.0 onto the FPU register stack.

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Push one of seven commonly used constants (in double extended-precision floating-point format) onto the FPU register stack. The constants that can be loaded with these instructions include +1.0, +0.0, $\log_2 10$, $\log_2 e$, π, $\log_{10} 2$, and $\log_e 2$. For each constant, an internal 66-bit constant is rounded (as specified by the RC field in the FPU control word) to double extended-precision floating-point format. The inexact-result exception (#P) is not generated as a result of the rounding, nor is the C1 flag set in the x87 FPU status word if the value is rounded up.

See the section titled "Approximation of Pi" in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for a description of the π constant.

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

IA-32 Architecture Compatibility

When the RC field is set to round-to-nearest, the FPU produces the same constants that is produced by the Intel 8087 and Intel 287 math coprocessors.

Operation

```
TOP := TOP – 1;
ST(0) := CONSTANT;
```

FPU Flags Affected

C1	Set to 1 if stack overflow occurred; otherwise, set to 0.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack overflow occurred.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FLDCW—Load x87 FPU Control Word

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 /5	FLDCW m2byte	Valid	Valid	Load FPU control word from m2byte.

Description

Loads the 16-bit source operand into the FPU control word. The source operand is a memory location. This instruction is typically used to establish or change the FPU's mode of operation.

If one or more exception flags are set in the FPU status word prior to loading a new FPU control word and the new control word unmasks one or more of those exceptions, a floating-point exception will be generated upon execution of the next floating-point instruction (except for the no-wait floating-point instructions, see the section titled "Software Exception Handling" in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1). To avoid raising exceptions when changing FPU operating modes, clear any pending exceptions (using the FCLEX or FNCLEX instruction) before loading the new control word.

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

Operation

```
FPUControlWord := SRC;
```

FPU Flags Affected

C0, C1, C2, C3 undefined.

Floating-Point Exceptions

None; however, this operation might unmask a pending exception in the FPU status word. That exception is then generated upon execution of the next "waiting" floating-point instruction.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FLDENV—Load x87 FPU Environment

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 /4	FLDENV m14/28byte	Valid	Valid	Load FPU environment from m14byte or m28byte.

Description

Loads the complete x87 FPU operating environment from memory into the FPU registers. The source operand specifies the first byte of the operating-environment data in memory. This data is typically written to the specified memory location by a FSTENV or FNSTENV instruction.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, show the layout in memory of the loaded environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used.

The FLDENV instruction should be executed in the same operating mode as the corresponding FSTENV/FNSTENV instruction.

If one or more unmasked exception flags are set in the new FPU status word, a floating-point exception will be generated upon execution of the next floating-point instruction (except for the no-wait floating-point instructions, see the section titled "Software Exception Handling" in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1). To avoid generating exceptions when loading a new environment, clear all the exception flags in the FPU status word that is being loaded.

If a page or limit fault occurs during the execution of this instruction, the state of the x87 FPU registers as seen by the fault handler may be different than the state being loaded from memory. In such situations, the fault handler should ignore the status of the x87 FPU registers, handle the fault, and return. The FLDENV instruction will then complete the loading of the x87 FPU registers with no resulting context inconsistency.

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

Operation

```
FPUControlWord := SRC[FPUControlWord];
FPUStatusWord := SRC[FPUStatusWord];
FPUTagWord := SRC[FPUTagWord];
FPUDataPointer := SRC[FPUDataPointer];
FPUInstructionPointer := SRC[FPUInstructionPointer];
FPULastInstructionOpcode := SRC[FPULastInstructionOpcode];
```

FPU Flags Affected

The C0, C1, C2, C3 flags are loaded.

Floating-Point Exceptions

None; however, if an unmasked exception is loaded in the status word, it is generated upon execution of the next "waiting" floating-point instruction.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FMUL/FMULP/FIMUL—Multiply

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D8 /1	FMUL m32fp	Valid	Valid	Multiply ST(0) by m32fp and store result in ST(0).
DC /1	FMUL m64fp	Valid	Valid	Multiply ST(0) by m64fp and store result in ST(0).
D8 C8+i	FMUL ST(0), ST(i)	Valid	Valid	Multiply ST(0) by ST(i) and store result in ST(0).
DC C8+i	FMUL ST(i), ST(0)	Valid	Valid	Multiply ST(i) by ST(0) and store result in ST(i).
DE C8+i	FMULP ST(i), ST(0)	Valid	Valid	Multiply ST(i) by ST(0), store result in ST(i), and pop the register stack.
DE C9	FMULP	Valid	Valid	Multiply ST(1) by ST(0), store result in ST(1), and pop the register stack.
DA /1	FIMUL m32int	Valid	Valid	Multiply ST(0) by m32int and store result in ST(0).
DE /1	FIMUL m16int	Valid	Valid	Multiply ST(0) by m16int and store result in ST(0).

Description

Multiplies the destination and source operands and stores the product in the destination location. The destination operand is always an FPU data register; the source operand can be an FPU data register or a memory location. Source operands in memory can be in single precision or double precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction multiplies the contents of the ST(1) register by the contents of the ST(0) register and stores the product in the ST(1) register. The one-operand version multiplies the contents of the ST(0) register by the contents of a memory location (either a floating-point or an integer value) and stores the product in the ST(0) register. The two-operand version, multiplies the contents of the ST(0) register by the contents of the ST(i) register, or vice versa, with the result being stored in the register specified with the first operand (the destination operand).

The FMULP instructions perform the additional operation of popping the FPU register stack after storing the product. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point multiply instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FMUL rather than FMULP.

The FIMUL instructions convert an integer source operand to double extended-precision floating-point format before performing the multiplication.

The sign of the result is always the exclusive-OR of the source signs, even if one or more of the values being multiplied is 0 or ∞. When the source operand is an integer 0, it is treated as a +0.

The following table shows the results obtained when multiplying various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-31. FMUL/FMULP/FIMUL Results

		DEST						
		−∞	−F	−0	+0	+F	+∞	NaN
SRC	−∞	+∞	+∞	*	*	−∞	−∞	NaN
	−F	+∞	+F	+0	−0	−F	−∞	NaN
	−I	+∞	+F	+0	−0	−F	−∞	NaN
	−0	*	+0	+0	−0	−0	*	NaN
	+0	*	−0	−0	+0	+0	*	NaN
	+I	−∞	−F	−0	+0	+F	+∞	NaN
	+F	−∞	−F	−0	+0	+F	+∞	NaN
	+∞	−∞	−∞	*	*	+∞	+∞	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means Integer.

* Indicates invalid-arithmetic-operand (#IA) exception.

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

Operation

```
IF Instruction = FIMUL
    THEN
        DEST := DEST * ConvertToDoubleExtendedPrecisionFP(SRC);
    ELSE (* Source operand is floating-point value *)
        DEST := DEST * SRC;
FI;
IF Instruction = FMULP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format.
	One operand is ±0 and the other is ±∞.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FNOP—No Operation

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 D0	FNOP	Valid	Valid	No operation is performed.

Description

Performs no FPU operation. This instruction takes up space in the instruction stream but does not affect the FPU or machine context, except the EIP register and the FPU Instruction Pointer.

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

FPU Flags Affected

C0, C1, C2, C3 undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FPATAN—Partial Arctangent

Opcode[1]	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F3	FPATAN	Valid	Valid	Replace ST(1) with arctan(ST(1)/ST(0)) and pop the register stack.

NOTES:

1. See IA-32 Architecture Compatibility section below.

Description

Computes the arctangent of the source operand in register ST(1) divided by the source operand in register ST(0), stores the result in ST(1), and pops the FPU register stack. The result in register ST(0) has the same sign as the source operand ST(1) and a magnitude less than +π.

The FPATAN instruction returns the angle between the X axis and the line from the origin to the point (X,Y), where Y (the ordinate) is ST(1) and X (the abscissa) is ST(0). The angle depends on the sign of X and Y independently, not just on the sign of the ratio Y/X. This is because a point (–X,Y) is in the second quadrant, resulting in an angle between π/2 and π, while a point (X,–Y) is in the fourth quadrant, resulting in an angle between 0 and –π/2. A point (–X,–Y) is in the third quadrant, giving an angle between –π/2 and –π.

The following table shows the results obtained when computing the arctangent of various classes of numbers, assuming that underflow does not occur.

Table 3-32. FPATAN Results

		ST(0)						
		– ∞	– F	– 0	+ 0	+ F	+ ∞	NaN
ST(1)	– ∞	– 3π/4*	– π/2	– π/2	– π/2	– π/2	– π/4*	NaN
	– F	-p	–π to –π/2	–π/2	–π/2	–π/2 to –0	- 0	NaN
	– 0	-p	-p	-p*	– 0*	– 0	– 0	NaN
	+ 0	+p	+ p	+ π*	+ 0*	+ 0	+ 0	NaN
	+ F	+p	+π to +π/2	+ π/2	+π/2	+π/2 to +0	+ 0	NaN
	+ ∞	+3π/4*	+π/2	+π/2	+π/2	+ π/2	+ π/4*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Table 8-10 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, specifies that the ratios 0/0 and ∞/∞ generate the floating-point invalid arithmetic-operation exception and, if this exception is masked, the floating-point QNaN indefinite value is returned. With the FPATAN instruction, the 0/0 or ∞/∞ value is actually not calculated using division. Instead, the arctangent of the two variables is derived from a standard mathematical formulation that is generalized to allow complex numbers as arguments. In this complex variable formulation, arctangent(0,0) etc. has well defined values. These values are needed to develop a library to compute transcendental functions with complex arguments, based on the FPU functions that only allow floating-point values as arguments.

There is no restriction on the range of source operands that FPATAN can accept.

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

IA-32 Architecture Compatibility

The source operands for this instruction are restricted for the 80287 math coprocessor to the following range:

$0 \le |ST(1)| < |ST(0)| < +\infty$

Operation

```
ST(1) := arctan(ST(1) / ST(0));
PopRegisterStack;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FPREM—Partial Remainder

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F8	FPREM	Valid	Valid	Replace ST(0) with the remainder obtained from dividing ST(0) by ST(1).

Description

Computes the remainder obtained from dividing the value in the ST(0) register (the dividend) by the value in the ST(1) register (the divisor or **modulus**), and stores the result in ST(0). The remainder represents the following value:

Remainder := ST(0) – (Q * ST(1))

Here, Q is an integer value that is obtained by truncating the floating-point number quotient of [ST(0) / ST(1)] toward zero. The sign of the remainder is the same as the sign of the dividend. The magnitude of the remainder is less than that of the modulus, unless a partial remainder was computed (as described below).

This instruction produces an exact result; the inexact-result exception does not occur and the rounding control has no effect. The following table shows the results obtained when computing the remainder of various classes of numbers, assuming that underflow does not occur.

Table 3-33. FPREM Results

		ST(1)						
		-∞	-F	-0	+0	+F	+∞	NaN
ST(0)	-∞	*	*	*	*	*	*	NaN
	-F	ST(0)	-F or -0	*	*	-F or -0	ST(0)	NaN
	-0	-0	-0	*	*	-0	-0	NaN
	+0	+0	+0	*	*	+0	+0	NaN
	+F	ST(0)	+F or +0	*	*	+F or +0	ST(0)	NaN
	+∞	*	*	*	*	*	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

When the result is 0, its sign is the same as that of the dividend. When the modulus is ∞, the result is equal to the value in ST(0).

The FPREM instruction does not compute the remainder specified in IEEE Std 754. The IEEE specified remainder can be computed with the FPREM1 instruction. The FPREM instruction is provided for compatibility with the Intel 8087 and Intel287 math coprocessors.

The FPREM instruction gets its name "partial remainder" because of the way it computes the remainder. This instruction arrives at a remainder through iterative subtraction. It can, however, reduce the exponent of ST(0) by no more than 63 in one execution of the instruction. If the instruction succeeds in producing a remainder that is less than the modulus, the operation is complete and the C2 flag in the FPU status word is cleared. Otherwise, C2 is set, and the result in ST(0) is called the **partial remainder**. The exponent of the partial remainder will be less than the exponent of the original dividend by at least 32. Software can re-execute the instruction (using the partial remainder in ST(0) as the dividend) until C2 is cleared. (Note that while executing such a remainder-computation loop, a higher-priority interrupting routine that needs the FPU can force a context switch in-between the instructions in the loop.)

An important use of the FPREM instruction is to reduce the arguments of periodic functions. When reduction is complete, the instruction stores the three least-significant bits of the quotient in the C3, C1, and C0 flags of the FPU

status word. This information is important in argument reduction for the tangent function (using a modulus of π/4), because it locates the original angle in the correct one of eight sectors of the unit circle.

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

Operation

```
D := exponent(ST(0)) – exponent(ST(1));

IF D < 64
    THEN
        Q := Integer(TruncateTowardZero(ST(0) / ST(1)));
        ST(0) := ST(0) – (ST(1) * Q);
        C2 := 0;
        C0, C3, C1 := LeastSignificantBits(Q); (* Q2, Q1, Q0 *)
    ELSE
        C2 := 1;
        N := An implementation-dependent number between 32 and 63;
        QQ := Integer(TruncateTowardZero((ST(0) / ST(1)) / 2^(D – N)));
        ST(0) := ST(0) – (ST(1) * QQ * 2^(D – N));
FI;
```

FPU Flags Affected

C0	Set to bit 2 (Q2) of the quotient.
C1	Set to 0 if stack underflow occurred; otherwise, set to least significant bit of quotient (Q0).
C2	Set to 0 if reduction complete; set to 1 if incomplete.
C3	Set to bit 1 (Q1) of the quotient.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, modulus is 0, dividend is ∞, or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FPREM1—Partial Remainder

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F5	FPREM1	Valid	Valid	Replace ST(0) with the IEEE remainder obtained from dividing ST(0) by ST(1).

Description

Computes the IEEE remainder obtained from dividing the value in the ST(0) register (the dividend) by the value in the ST(1) register (the divisor or **modulus**), and stores the result in ST(0). The remainder represents the following value:

Remainder := ST(0) – (Q * ST(1))

Here, Q is an integer value that is obtained by rounding the floating-point number quotient of [ST(0) / ST(1)] toward the nearest integer value. The magnitude of the remainder is less than or equal to half the magnitude of the modulus, unless a partial remainder was computed (as described below).

This instruction produces an exact result; the precision (inexact) exception does not occur and the rounding control has no effect. The following table shows the results obtained when computing the remainder of various classes of numbers, assuming that underflow does not occur.

Table 3-34. FPREM1 Results

		ST(1)						
		– ∞	– F	– 0	+ 0	+ F	+ ∞	NaN
ST(0)	– ∞	*	*	*	*	*	*	NaN
	– F	ST(0)	±F or –0	*	*	± F or – 0	ST(0)	NaN
	– 0	– 0	– 0	*	*	– 0	-0	NaN
	+ 0	+ 0	+ 0	*	*	+ 0	+0	NaN
	+ F	ST(0)	± F or + 0	*	*	± F or + 0	ST(0)	NaN
	+ ∞	*	*	*	*	*	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

When the result is 0, its sign is the same as that of the dividend. When the modulus is ∞, the result is equal to the value in ST(0).

The FPREM1 instruction computes the remainder specified in IEEE Standard 754. This instruction operates differently from the FPREM instruction in the way that it rounds the quotient of ST(0) divided by ST(1) to an integer (see the "Operation" section below).

Like the FPREM instruction, FPREM1 computes the remainder through iterative subtraction, but can reduce the exponent of ST(0) by no more than 63 in one execution of the instruction. If the instruction succeeds in producing a remainder that is less than one half the modulus, the operation is complete and the C2 flag in the FPU status word is cleared. Otherwise, C2 is set, and the result in ST(0) is called the **partial remainder**. The exponent of the partial remainder will be less than the exponent of the original dividend by at least 32. Software can re-execute the instruction (using the partial remainder in ST(0) as the dividend) until C2 is cleared. (Note that while executing such a remainder-computation loop, a higher-priority interrupting routine that needs the FPU can force a context switch in-between the instructions in the loop.)

An important use of the FPREM1 instruction is to reduce the arguments of periodic functions. When reduction is complete, the instruction stores the three least-significant bits of the quotient in the C3, C1, and C0 flags of the FPU status word. This information is important in argument reduction for the tangent function (using a modulus of π/4), because it locates the original angle in the correct one of eight sectors of the unit circle.

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

Operation

```
D := exponent(ST(0)) – exponent(ST(1));
IF D < 64
    THEN
        Q := Integer(RoundTowardNearestInteger(ST(0) / ST(1)));
        ST(0) := ST(0) – (ST(1) * Q);
        C2 := 0;
        C0, C3, C1 := LeastSignificantBits(Q); (* Q2, Q1, Q0 *)
    ELSE
        C2 := 1;
        N := An implementation-dependent number between 32 and 63;
        QQ := Integer(TruncateTowardZero((ST(0) / ST(1)) / 2^(D – N)));
        ST(0) := ST(0) – (ST(1) * QQ * 2^(D – N));
FI;
```

FPU Flags Affected

C0	Set to bit 2 (Q2) of the quotient.
C1	Set to 0 if stack underflow occurred; otherwise, set to least significant bit of quotient (Q0).
C2	Set to 0 if reduction complete; set to 1 if incomplete.
C3	Set to bit 1 (Q1) of the quotient.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, modulus (divisor) is 0, dividend is ∞, or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FPTAN—Partial Tangent

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F2	FPTAN	Valid	Valid	Replace ST(0) with its approximate tangent and push 1 onto the FPU stack.

Description

Computes the approximate tangent of the source operand in register ST(0), stores the result in ST(0), and pushes a 1.0 onto the FPU register stack. The source operand must be given in radians and must be less than $\pm 2^{63}$. The following table shows the unmasked results obtained when computing the partial tangent of various classes of numbers, assuming that underflow does not occur.

Table 3-35. FPTAN Results

ST(0) SRC	ST(0) DEST
$-\infty$	*
−F	−F to +F
−0	-0
+0	+0
+F	−F to +F
$+\infty$	*
NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range -2^{63} to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of 2π. However, even within the range -2^{63} to $+2^{63}$, inaccurate results can occur because the finite approximation of π used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FPTAN only to arguments reduced accurately in software, to a value smaller in absolute value than $3\pi/8$. See the sections titled "Approximation of Pi" and "Transcendental Instruction Accuracy" in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for a discussion of the proper value to use for π in performing such reductions.

The value 1.0 is pushed onto the register stack after the tangent has been computed to maintain compatibility with the Intel 8087 and Intel287 math coprocessors. This operation also simplifies the calculation of other trigonometric functions. For instance, the cotangent (which is the reciprocal of the tangent) can be computed by executing a FDIVR instruction after the FPTAN instruction.

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

Operation

```
IF ST(0) < 2^63
    THEN
        C2 := 0;
        ST(0) := fptan(ST(0)); // approximation of tan
        TOP := TOP – 1;
        ST(0) := 1.0;
    ELSE (* Source operand is out-of-range *)
        C2 := 1;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; set to 1 if stack overflow occurred. Set if result was rounded up; cleared otherwise.
C2	Set to 1 if outside range (-2^{63} < source operand < $+2^{63}$); otherwise, set to 0.
C0, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow or overflow occurred.
#IA	Source operand is an SNaN value, ∞, or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FRNDINT—Round to Integer

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 FC	FRNDINT	Valid	Valid	Round ST(0) to an integer.

Description

Rounds the source value in the ST(0) register to the nearest integral value, depending on the current rounding mode (setting of the RC field of the FPU control word), and stores the result in ST(0).

If the source value is ∞, the value is not changed. If the source value is not an integral value, the floating-point inexact-result exception (#P) is generated.

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

Operation

```
ST(0) := RoundToIntegralValue(ST(0));
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#P	Source operand is not an integral value.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FRSTOR—Restore x87 FPU State

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DD /4	FRSTOR m94/108byte	Valid	Valid	Load FPU state from m94byte or m108byte.

Description

Loads the FPU state (operating environment and register stack) from the memory area specified with the source operand. This state data is typically written to the specified memory location by a previous FSAVE/FNSAVE instruction.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used. The contents of the FPU register stack are stored in the 80 bytes immediately following the operating environment image.

The FRSTOR instruction should be executed in the same operating mode as the corresponding FSAVE/FNSAVE instruction.

If one or more unmasked exception bits are set in the new FPU status word, a floating-point exception will be generated upon execution of the next floating-point instruction (except for the no-wait floating-point instructions, see the section titled "Software Exception Handling" in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1). To avoid raising exceptions when loading a new operating environment, clear all the exception flags in the FPU status word that is being loaded.

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

Operation

```
FPUControlWord := SRC[FPUControlWord];
FPUStatusWord := SRC[FPUStatusWord];
FPUTagWord := SRC[FPUTagWord];
FPUDataPointer := SRC[FPUDataPointer];
FPUInstructionPointer := SRC[FPUInstructionPointer];
FPULastInstructionOpcode := SRC[FPULastInstructionOpcode];

ST(0) := SRC[ST(0)];
ST(1) := SRC[ST(1)];
ST(2) := SRC[ST(2)];
ST(3) := SRC[ST(3)];
ST(4) := SRC[ST(4)];
ST(5) := SRC[ST(5)];
ST(6) := SRC[ST(6)];
ST(7) := SRC[ST(7)];
```

FPU Flags Affected

The C0, C1, C2, C3 flags are loaded.

Floating-Point Exceptions

None; however, if an unmasked exception is loaded in the status word, it is generated upon execution of the next "waiting" floating-point instruction.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FSAVE/FNSAVE—Store x87 FPU State

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
9B DD /6	FSAVE m94/108byte	Valid	Valid	Store FPU state to m94byte or m108byte after checking for pending unmasked floating-point exceptions. Then re-initialize the FPU.
DD /6	FNSAVE[1] m94/108byte	Valid	Valid	Store FPU environment to m94byte or m108byte without checking for pending unmasked floating-point exceptions. Then re-initialize the FPU.

NOTES:

1. See IA-32 Architecture Compatibility section below.

Description

Stores the current FPU state (operating environment and register stack) at the specified destination in memory, and then re-initializes the FPU. The FSAVE instruction checks for and handles pending unmasked floating-point exceptions before storing the FPU state; the FNSAVE instruction does not.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used. The contents of the FPU register stack are stored in the 80 bytes immediately follow the operating environment image.

The saved image reflects the state of the FPU after all floating-point instructions preceding the FSAVE/FNSAVE instruction in the instruction stream have been executed.

After the FPU state has been saved, the FPU is reset to the same default values it is set to with the FINIT/FNINIT instructions (see "FINIT/FNINIT—Initialize Floating-Point Unit" in this chapter).

The FSAVE/FNSAVE instructions are typically used when the operating system needs to perform a context switch, an exception handler needs to use the FPU, or an application program needs to pass a "clean" FPU to a procedure.

The assembler issues two instructions for the FSAVE instruction (an FWAIT instruction followed by an FNSAVE instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

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

IA-32 Architecture Compatibility

For Intel math coprocessors and FPUs prior to the Intel Pentium processor, an FWAIT instruction should be executed before attempting to read from the memory image stored with a prior FSAVE/FNSAVE instruction. This FWAIT instruction helps ensure that the storage operation has been completed.

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSAVE instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for a description of these circumstances. An FNSAVE instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

Operation

```
(* Save FPU State and Registers *)

DEST[FPUControlWord] := FPUControlWord;
DEST[FPUStatusWord] := FPUStatusWord;
DEST[FPUTagWord] := FPUTagWord;
DEST[FPUDataPointer] := FPUDataPointer;
DEST[FPUInstructionPointer] := FPUInstructionPointer;
DEST[FPULastInstructionOpcode] := FPULastInstructionOpcode;

DEST[ST(0)] := ST(0);
DEST[ST(1)] := ST(1);
DEST[ST(2)] := ST(2);
DEST[ST(3)] := ST(3);
DEST[ST(4)]:= ST(4);
DEST[ST(5)] := ST(5);
DEST[ST(6)] := ST(6);
DEST[ST(7)] := ST(7);

(* Initialize FPU *)

FPUControlWord := 037FH;
FPUStatusWord := 0;
FPUTagWord := FFFFH;
FPUDataPointer := 0;
FPUInstructionPointer := 0;
FPULastInstructionOpcode := 0;
```

FPU Flags Affected

The C0, C1, C2, and C3 flags are saved and then cleared.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	If destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

FSCALE—Scale

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 FD	FSCALE	Valid	Valid	Scale ST(0) by ST(1).

Description

Truncates the value in the source operand (toward 0) to an integral value and adds that value to the exponent of the destination operand. The destination and source operands are floating-point values located in registers ST(0) and ST(1), respectively. This instruction provides rapid multiplication or division by integral powers of 2. The following table shows the results obtained when scaling various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-36. FSCALE Results

	ST(1)							
		−∞	−F	−0	+0	+F	+∞	NaN
ST(0)	−∞	NaN	−∞	−∞	−∞	−∞	−∞	NaN
	−F	−0	−F	−F	−F	−F	−∞	NaN
	−0	−0	−0	−0	−0	−0	NaN	NaN
	+0	+0	+0	+0	+0	+0	NaN	NaN
	+F	+0	+F	+F	+F	+F	+∞	NaN
	+∞	NaN	+∞	+∞	+∞	+∞	+∞	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

In most cases, only the exponent is changed and the mantissa (significand) remains unchanged. However, when the value being scaled in ST(0) is a denormal value, the mantissa is also changed and the result may turn out to be a normalized number. Similarly, if overflow or underflow results from a scale operation, the resulting mantissa will differ from the source's mantissa.

The FSCALE instruction can also be used to reverse the action of the FXTRACT instruction, as shown in the following example:

```
FXTRACT;
FSCALE;
FSTP ST(1);
```

In this example, the FXTRACT instruction extracts the significand and exponent from the value in ST(0) and stores them in ST(0) and ST(1) respectively. The FSCALE then scales the significand in ST(0) by the exponent in ST(1), recreating the original value before the FXTRACT operation was performed. The FSTP ST(1) instruction overwrites the exponent (extracted by the FXTRACT instruction) with the recreated value, which returns the stack to its original state with only one register [ST(0)] occupied.

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

Operation

$ST(0) := ST(0) * 2^{RoundTowardZero(ST(1))};$

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FSIN—Sine

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 FE	FSIN	Valid	Valid	Replace ST(0) with the approximate of its sine.

Description

Computes an approximation of the sine of the source operand in register ST(0) and stores the result in ST(0). The source operand must be given in radians and must be within the range -2^{63} to $+2^{63}$. The following table shows the results obtained when taking the sine of various classes of numbers, assuming that underflow does not occur.

Table 3-37. FSIN Results

SRC (ST(0))	DEST (ST(0))
− ∞	*
− F	− 1 to + 1
− 0	−0
+ 0	+ 0
+ F	− 1 to +1
+ ∞	*
NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range -2^{63} to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of 2π. However, even within the range -2^{63} to $+2^{63}$, inaccurate results can occur because the finite approximation of π used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FSIN only to arguments reduced accurately in software, to a value smaller in absolute value than $3\pi/4$. See the sections titled "Approximation of Pi" and "Transcendental Instruction Accuracy" in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for a discussion of the proper value to use for π in performing such reductions.

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

Operation

```
IF -2^63 < ST(0) < 2^63
    THEN
        C2 := 0;
        ST(0) := fsin(ST(0)); // approximation of the mathematical sin function
    ELSE (* Source operand out of range *)
        C2 := 1;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C2	Set to 1 if outside range (-2^{63} < source operand < $+2^{63}$); otherwise, set to 0.
C0, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, ∞, or unsupported format.
#D	Source operand is a denormal value.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FSINCOS—Sine and Cosine

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 FB	FSINCOS	Valid	Valid	Compute the sine and cosine of ST(0); replace ST(0) with the approximate sine, and push the approximate cosine onto the register stack.

Description

Computes both the approximate sine and the cosine of the source operand in register ST(0), stores the sine in ST(0), and pushes the cosine onto the top of the FPU register stack. (This instruction is faster than executing the FSIN and FCOS instructions in succession.)

The source operand must be given in radians and must be within the range -2^{63} to $+2^{63}$. The following table shows the results obtained when taking the sine and cosine of various classes of numbers, assuming that underflow does not occur.

Table 3-38. FSINCOS Results

SRC	DEST	
ST(0)	ST(1) Cosine	ST(0) Sine
$-\infty$	*	*
−F	−1 to +1	−1 to +1
−0	+1	−0
+0	+1	+0
+F	−1 to +1	−1 to +1
$+\infty$	*	*
NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range -2^{63} to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of 2π. However, even within the range -2^{63} to $+2^{63}$, inaccurate results can occur because the finite approximation of π used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FSINCOS only to arguments reduced accurately in software, to a value smaller in absolute value than $3\pi/8$. See the sections titled "Approximation of Pi" and "Transcendental Instruction Accuracy" in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for a discussion of the proper value to use for π in performing such reductions.

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

Operation

```
IF ST(0) < 2^63
    THEN
        C2 := 0;
        TEMP := fcos(ST(0)); // approximation of cosine
        ST(0) := fsin(ST(0)); // approximation of sine
        TOP := TOP – 1;
        ST(0) := TEMP;
    ELSE (* Source operand out of range *)
        C2 := 1;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; set to 1 of stack overflow occurs. Set if result was rounded up; cleared otherwise.
C2	Set to 1 if outside range (-2^{63} < source operand < $+2^{63}$); otherwise, set to 0.
C0, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow or overflow occurred.
#IA	Source operand is an SNaN value, ∞, or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FSQRT—Square Root

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 FA	FSQRT	Valid	Valid	Computes square root of ST(0) and stores the result in ST(0).

Description

Computes the square root of the source value in the ST(0) register and stores the result in ST(0).

The following table shows the results obtained when taking the square root of various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-39. FSQRT Results

SRC (ST(0))	DEST (ST(0))
$-\infty$	*
−F	*
−0	−0
+0	+0
+F	+F
$+\infty$	$+\infty$
NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

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

Operation

```
ST(0) := SquareRoot(ST(0));
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
	Source operand is a negative value (except for −0).
#D	Source operand is a denormal value.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FST/FSTP—Store Floating-Point Value

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 /2	FST m32fp	Valid	Valid	Copy ST(0) to m32fp.
DD /2	FST m64fp	Valid	Valid	Copy ST(0) to m64fp.
DD D0+i	FST ST(i)	Valid	Valid	Copy ST(0) to ST(i).
D9 /3	FSTP m32fp	Valid	Valid	Copy ST(0) to m32fp and pop register stack.
DD /3	FSTP m64fp	Valid	Valid	Copy ST(0) to m64fp and pop register stack.
DB /7	FSTP m80fp	Valid	Valid	Copy ST(0) to m80fp and pop register stack.
DD D8+i	FSTP ST(i)	Valid	Valid	Copy ST(0) to ST(i) and pop register stack.

Description

The FST instruction copies the value in the ST(0) register to the destination operand, which can be a memory location or another register in the FPU register stack. When storing the value in memory, the value is converted to single precision or double precision floating-point format.

The FSTP instruction performs the same operation as the FST instruction and then pops the register stack. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The FSTP instruction can also store values in memory in double extended-precision floating-point format.

If the destination operand is a memory location, the operand specifies the address where the first byte of the destination value is to be stored. If the destination operand is a register, the operand specifies a register in the register stack relative to the top of the stack.

If the destination size is single precision or double precision, the significand of the value being stored is rounded to the width of the destination (according to the rounding mode specified by the RC field of the FPU control word), and the exponent is converted to the width and bias of the destination format. If the value being stored is too large for the destination format, a numeric overflow exception (#O) is generated and, if the exception is unmasked, no value is stored in the destination operand. If the value being stored is a denormal value, the denormal exception (#D) is not generated. This condition is simply signaled as a numeric underflow exception (#U) condition.

If the value being stored is ±0, ±∞, or a NaN, the least-significant bits of the significand and the exponent are truncated to fit the destination format. This operation preserves the value's identity as a 0, ∞, or NaN.

If the destination operand is a non-empty register, the invalid-operation exception is not generated.

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

Operation

```
DEST := ST(0);

IF Instruction = FSTP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Indicates rounding direction of if the floating-point inexact exception (#P) is generated: 0 := not roundup; 1 := roundup.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	If destination result is an SNaN value or unsupported format, except when the destination format is in double extended-precision floating-point format.
#U	Result is too small for the destination format.
#O	Result is too large for the destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FSTCW/FNSTCW—Store x87 FPU Control Word

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
9B D9 /7	FSTCW m2byte	Valid	Valid	Store FPU control word to m2byte after checking for pending unmasked floating-point exceptions.
D9 /7	FNSTCW[1] m2byte	Valid	Valid	Store FPU control word to m2byte without checking for pending unmasked floating-point exceptions.

NOTES:

1. See IA-32 Architecture Compatibility section below.

Description

Stores the current value of the FPU control word at the specified destination in memory. The FSTCW instruction checks for and handles pending unmasked floating-point exceptions before storing the control word; the FNSTCW instruction does not.

The assembler issues two instructions for the FSTCW instruction (an FWAIT instruction followed by an FNSTCW instruction), and the processor executes each of these instructions in separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

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

IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTCW instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for a description of these circumstances. An FNSTCW instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

Operation

```
DEST := FPUControlWord;
```

FPU Flags Affected

The C0, C1, C2, and C3 flags are undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FSTENV/FNSTENV—Store x87 FPU Environment

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
9B D9 /6	FSTENV m14/28byte	Valid	Valid	Store FPU environment to m14byte or m28byte after checking for pending unmasked floating-point exceptions. Then mask all floating-point exceptions.
D9 /6	FNSTENV[1] m14/28byte	Valid	Valid	Store FPU environment to m14byte or m28byte without checking for pending unmasked floating-point exceptions. Then mask all floating-point exceptions.

NOTES:

1. See IA-32 Architecture Compatibility section below.

Description

Saves the current FPU operating environment at the memory location specified with the destination operand, and then masks all floating-point exceptions. The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used.

The FSTENV instruction checks for and handles any pending unmasked floating-point exceptions before storing the FPU environment; the FNSTENV instruction does not. The saved image reflects the state of the FPU after all floating-point instructions preceding the FSTENV/FNSTENV instruction in the instruction stream have been executed.

These instructions are often used by exception handlers because they provide access to the FPU instruction and data pointers. The environment is typically saved in the stack. Masking all exceptions after saving the environment prevents floating-point exceptions from interrupting the exception handler.

The assembler issues two instructions for the FSTENV instruction (an FWAIT instruction followed by an FNSTENV instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

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

IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTENV instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for a description of these circumstances. An FNSTENV instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

Operation

```
DEST[FPUControlWord] := FPUControlWord;
DEST[FPUStatusWord] := FPUStatusWord;
DEST[FPUTagWord] := FPUTagWord;
DEST[FPUDataPointer] := FPUDataPointer;
DEST[FPUInstructionPointer] := FPUInstructionPointer;
DEST[FPULastInstructionOpcode] := FPULastInstructionOpcode;
```

FPU Flags Affected

The C0, C1, C2, and C3 are undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FSTSW/FNSTSW—Store x87 FPU Status Word

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
9B DD /7	FSTSW m2byte	Valid	Valid	Store FPU status word at m2byte after checking for pending unmasked floating-point exceptions.
9B DF E0	FSTSW AX	Valid	Valid	Store FPU status word in AX register after checking for pending unmasked floating-point exceptions.
DD /7	FNSTSW[1] m2byte	Valid	Valid	Store FPU status word at m2byte without checking for pending unmasked floating-point exceptions.
DF E0	FNSTSW[1] AX	Valid	Valid	Store FPU status word in AX register without checking for pending unmasked floating-point exceptions.

NOTES:

1. See IA-32 Architecture Compatibility section below.

Description

Stores the current value of the x87 FPU status word in the destination location. The destination operand can be either a two-byte memory location or the AX register. The FSTSW instruction checks for and handles pending unmasked floating-point exceptions before storing the status word; the FNSTSW instruction does not.

The FNSTSW AX form of the instruction is used primarily in conditional branching (for instance, after an FPU comparison instruction or an FPREM, FPREM1, or FXAM instruction), where the direction of the branch depends on the state of the FPU condition code flags. (See the section titled "Branching and Conditional Moves on FPU Condition Codes" in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.) This instruction can also be used to invoke exception handlers (by examining the exception flags) in environments that do not use interrupts. When the FNSTSW AX instruction is executed, the AX register is updated before the processor executes any further instructions. The status stored in the AX register is thus guaranteed to be from the completion of the prior FPU instruction.

The assembler issues two instructions for the FSTSW instruction (an FWAIT instruction followed by an FNSTSW instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

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

IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTSW instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for a description of these circumstances. An FNSTSW instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

Operation

```
DEST := FPUStatusWord;
```

FPU Flags Affected

The C0, C1, C2, and C3 are undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FSUB/FSUBP/FISUB—Subtract

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D8 /4	FSUB m32fp	Valid	Valid	Subtract m32fp from ST(0) and store result in ST(0).
DC /4	FSUB m64fp	Valid	Valid	Subtract m64fp from ST(0) and store result in ST(0).
D8 E0+i	FSUB ST(0), ST(i)	Valid	Valid	Subtract ST(i) from ST(0) and store result in ST(0).
DC E8+i	FSUB ST(i), ST(0)	Valid	Valid	Subtract ST(0) from ST(i) and store result in ST(i).
DE E8+i	FSUBP ST(i), ST(0)	Valid	Valid	Subtract ST(0) from ST(i), store result in ST(i), and pop register stack.
DE E9	FSUBP	Valid	Valid	Subtract ST(0) from ST(1), store result in ST(1), and pop register stack.
DA /4	FISUB m32int	Valid	Valid	Subtract m32int from ST(0) and store result in ST(0).
DE /4	FISUB m16int	Valid	Valid	Subtract m16int from ST(0) and store result in ST(0).

Description

Subtracts the source operand from the destination operand and stores the difference in the destination location. The destination operand is always an FPU data register; the source operand can be a register or a memory location. Source operands in memory can be in single precision or double precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction subtracts the contents of the ST(0) register from the ST(1) register and stores the result in ST(1). The one-operand version subtracts the contents of a memory location (either a floating-point or an integer value) from the contents of the ST(0) register and stores the result in ST(0). The two-operand version, subtracts the contents of the ST(0) register from the ST(i) register or vice versa.

The FSUBP instructions perform the additional operation of popping the FPU register stack following the subtraction. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point subtract instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FSUB rather than FSUBP.

The FISUB instructions convert an integer source operand to double extended-precision floating-point format before performing the subtraction.

Table 3-40 shows the results obtained when subtracting various classes of numbers from one another, assuming that neither overflow nor underflow occurs. Here, the SRC value is subtracted from the DEST value (DEST – SRC = result).

When the difference between two operands of like sign is 0, the result is +0, except for the round toward –∞ mode, in which case the result is –0. This instruction also guarantees that +0 – (–0) = +0, and that –0 – (+0) = –0. When the source operand is an integer 0, it is treated as a +0.

When one operand is ∞, the result is ∞ of the expected sign. If both operands are ∞ of the same sign, an invalid-operation exception is generated.

Table 3-40. FSUB/FSUBP/FISUB Results

		SRC						
		– ∞	– F or – I	– 0	+ 0	+ F or + I	+ ∞	NaN
DEST	– ∞	*	– ∞	– ∞	– ∞	– ∞	– ∞	NaN
	– F	+ ∞	±F or ±0	DEST	DEST	– F	– ∞	NaN
	– 0	+ ∞	–SRC	±0	– 0	– SRC	– ∞	NaN
	+ 0	+ ∞	–SRC	+ 0	±0	– SRC	– ∞	NaN
	+ F	+ ∞	+ F	DEST	DEST	±F or ±0	– ∞	NaN
	+ ∞	+ ∞	+ ∞	+ ∞	+ ∞	+ ∞	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

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

Operation

```
IF Instruction = FISUB
    THEN
        DEST := DEST – ConvertToDoubleExtendedPrecisionFP(SRC);
    ELSE (* Source operand is floating-point value *)
        DEST := DEST – SRC;
FI;
IF Instruction = FSUBP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format.
	Operands are infinities of like sign.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FSUBR/FSUBRP/FISUBR—Reverse Subtract

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D8 /5	FSUBR m32fp	Valid	Valid	Subtract ST(0) from m32fp and store result in ST(0).
DC /5	FSUBR m64fp	Valid	Valid	Subtract ST(0) from m64fp and store result in ST(0).
D8 E8+i	FSUBR ST(0), ST(i)	Valid	Valid	Subtract ST(0) from ST(i) and store result in ST(0).
DC E0+i	FSUBR ST(i), ST(0)	Valid	Valid	Subtract ST(i) from ST(0) and store result in ST(i).
DE E0+i	FSUBRP ST(i), ST(0)	Valid	Valid	Subtract ST(i) from ST(0), store result in ST(i), and pop register stack.
DE E1	FSUBRP	Valid	Valid	Subtract ST(1) from ST(0), store result in ST(1), and pop register stack.
DA /5	FISUBR m32int	Valid	Valid	Subtract ST(0) from m32int and store result in ST(0).
DE /5	FISUBR m16int	Valid	Valid	Subtract ST(0) from m16int and store result in ST(0).

Description

Subtracts the destination operand from the source operand and stores the difference in the destination location. The destination operand is always an FPU register; the source operand can be a register or a memory location. Source operands in memory can be in single precision or double precision floating-point format or in word or doubleword integer format.

These instructions perform the reverse operations of the FSUB, FSUBP, and FISUB instructions. They are provided to support more efficient coding.

The no-operand version of the instruction subtracts the contents of the ST(1) register from the ST(0) register and stores the result in ST(1). The one-operand version subtracts the contents of the ST(0) register from the contents of a memory location (either a floating-point or an integer value) and stores the result in ST(0). The two-operand version, subtracts the contents of the ST(i) register from the ST(0) register or vice versa.

The FSUBRP instructions perform the additional operation of popping the FPU register stack following the subtraction. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point reverse subtract instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FSUBR rather than FSUBRP.

The FISUBR instructions convert an integer source operand to double extended-precision floating-point format before performing the subtraction.

The following table shows the results obtained when subtracting various classes of numbers from one another, assuming that neither overflow nor underflow occurs. Here, the DEST value is subtracted from the SRC value (SRC – DEST = result).

When the difference between two operands of like sign is 0, the result is +0, except for the round toward $-\infty$ mode, in which case the result is –0. This instruction also guarantees that +0 – (–0) = +0, and that –0 – (+0) = –0. When the source operand is an integer 0, it is treated as a +0.

When one operand is ∞, the result is ∞ of the expected sign. If both operands are ∞ of the same sign, an invalid-operation exception is generated.

Table 3-41. FSUBR/FSUBRP/FISUBR Results

		SRC						
		– ∞	–F or –I	–0	+0	+F or +I	+ ∞	NaN
DEST	– ∞	*	+ ∞	+ ∞	+ ∞	+ ∞	+ ∞	NaN
	– F	– ∞	±F or ±0	–DEST	–DEST	+ F	+ ∞	NaN
	– 0	– ∞	SRC	±0	+ 0	SRC	+ ∞	NaN
	+ 0	– ∞	SRC	– 0	±0	SRC	+ ∞	NaN
	+ F	– ∞	– F	–DEST	–DEST	±F or ±0	+ ∞	NaN
	+ ∞	– ∞	– ∞	– ∞	– ∞	– ∞	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

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

Operation

```
IF Instruction = FISUBR
    THEN
        DEST := ConvertToDoubleExtendedPrecisionFP(SRC) – DEST;
    ELSE (* Source operand is floating-point value *)
        DEST := SRC – DEST; FI;
IF Instruction = FSUBRP
    THEN
        PopRegisterStack; FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format.
	Operands are infinities of like sign.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FTST—TEST

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 E4	FTST	Valid	Valid	Compare ST(0) with 0.0.

Description

Compares the value in the ST(0) register with 0.0 and sets the condition code flags C0, C2, and C3 in the FPU status word according to the results (see table below).

Table 3-42. FTST Results

Condition	C3	C2	C0
ST(0) > 0.0	0	0	0
ST(0) < 0.0	0	0	1
ST(0) = 0.0	1	0	0
Unordered	1	1	1

This instruction performs an "unordered comparison." An unordered comparison also checks the class of the numbers being compared (see "FXAM—Examine Floating-Point" in this chapter). If the value in register ST(0) is a NaN or is in an undefined format, the condition flags are set to "unordered" and the invalid operation exception is generated.

The sign of zero is ignored, so that (– 0.0 := +0.0).

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

Operation

```
CASE (relation of operands) OF
    Not comparable:  C3, C2, C0 := 111;
    ST(0) > 0.0:     C3, C2, C0 := 000;
    ST(0) < 0.0:     C3, C2, C0 := 001;
    ST(0) = 0.0:     C3, C2, C0 := 100;
ESAC;
```

FPU Flags Affected

C1	Set to 0.
C0, C2, C3	See Table 3-42.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	The source operand is a NaN value or is in an unsupported format.
#D	The source operand is a denormal value.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FUCOM/FUCOMP/FUCOMPP—Unordered Compare Floating-Point Values

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DD E0+i	FUCOM ST(i)	Valid	Valid	Compare ST(0) with ST(i).
DD E1	FUCOM	Valid	Valid	Compare ST(0) with ST(1).
DD E8+i	FUCOMP ST(i)	Valid	Valid	Compare ST(0) with ST(i) and pop register stack.
DD E9	FUCOMP	Valid	Valid	Compare ST(0) with ST(1) and pop register stack.
DA E9	FUCOMPP	Valid	Valid	Compare ST(0) with ST(1) and pop register stack twice.

Description

Performs an unordered comparison of the contents of register ST(0) and ST(i) and sets condition code flags C0, C2, and C3 in the FPU status word according to the results (see the table below). If no operand is specified, the contents of registers ST(0) and ST(1) are compared. The sign of zero is ignored, so that −0.0 is equal to +0.0.

Table 3-43. FUCOM/FUCOMP/FUCOMPP Results

Comparison Results*	C3	C2	C0
ST0 > ST(i)	0	0	0
ST0 < ST(i)	0	0	1
ST0 = ST(i)	1	0	0
Unordered	1	1	1

NOTES:

* Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

An unordered comparison checks the class of the numbers being compared (see "FXAM—Examine Floating-Point" in this chapter). The FUCOM/FUCOMP/FUCOMPP instructions perform the same operations as the FCOM/FCOMP/FCOMPP instructions. The only difference is that the FUCOM/FUCOMP/FUCOMPP instructions raise the invalid-arithmetic-operand exception (#IA) only when either or both operands are an SNaN or are in an unsupported format; QNaNs cause the condition code flags to be set to unordered, but do not cause an exception to be generated. The FCOM/FCOMP/FCOMPP instructions raise an invalid-operation exception when either or both of the operands are a NaN value of any kind or are in an unsupported format.

As with the FCOM/FCOMP/FCOMPP instructions, if the operation results in an invalid-arithmetic-operand exception being raised, the condition code flags are set only if the exception is masked.

The FUCOMP instruction pops the register stack following the comparison operation and the FUCOMPP instruction pops the register stack twice following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

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

Operation

```
CASE (relation of operands) OF
    ST > SRC:       C3, C2, C0 := 000;
    ST < SRC:       C3, C2, C0 := 001;
    ST = SRC:       C3, C2, C0 := 100;
ESAC;

IF ST(0) or SRC = QNaN, but not SNaN or unsupported format
    THEN
        C3, C2, C0 := 111;
    ELSE (* ST(0) or SRC is SNaN or unsupported format *)
        #IA;
        IF FPUControlWord.IM = 1
            THEN
                C3, C2, C0 := 111;
        FI;
FI;

IF Instruction = FUCOMP
    THEN
        PopRegisterStack;
FI;

IF Instruction = FUCOMPP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
C0, C2, C3	See Table 3-43.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	One or both operands are SNaN values or have unsupported formats. Detection of a QNaN value in and of itself does not raise an invalid-operand exception.
#D	One or both operands are denormal values.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FXAM—Examine Floating-Point

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 E5	FXAM	Valid	Valid	Classify value or number in ST(0).

Description

Examines the contents of the ST(0) register and sets the condition code flags C0, C2, and C3 in the FPU status word to indicate the class of value or number in the register (see the table below).

Table 3-44. FXAM Results

Class	C3	C2	C0
Unsupported	0	0	0
NaN	0	0	1
Normal finite number	0	1	0
Infinity	0	1	1
Zero	1	0	0
Empty	1	0	1
Denormal number	1	1	0

The C1 flag is set to the sign of the value in ST(0), regardless of whether the register is empty or full.

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

Operation

```
C1 := sign bit of ST; (* 0 for positive, 1 for negative *)
CASE (class of value or number in ST(0)) OF
    Unsupported:C3, C2, C0 := 000;
    NaN:        C3, C2, C0 := 001;
    Normal:     C3, C2, C0 := 010;
    Infinity:   C3, C2, C0 := 011;
    Zero:       C3, C2, C0 := 100;
    Empty:      C3, C2, C0 := 101;
    Denormal:   C3, C2, C0 := 110;
ESAC;
```

FPU Flags Affected

C1	Sign of value in ST(0).
C0, C2, C3	See Table 3-44.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FXCH—Exchange Register Contents

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 C8+i	FXCH ST(i)	Valid	Valid	Exchange the contents of ST(0) and ST(i).
D9 C9	FXCH	Valid	Valid	Exchange the contents of ST(0) and ST(1).

Description

Exchanges the contents of registers ST(0) and ST(i). If no source operand is specified, the contents of ST(0) and ST(1) are exchanged.

This instruction provides a simple means of moving values in the FPU register stack to the top of the stack [ST(0)], so that they can be operated on by those floating-point instructions that can only operate on values in ST(0). For example, the following instruction sequence takes the square root of the third register from the top of the register stack:

```
FXCH ST(3);
FSQRT;
FXCH ST(3);
```

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

Operation

```
IF (Number-of-operands) is 1
    THEN
        temp := ST(0);
        ST(0) := SRC;
        SRC := temp;
    ELSE
        temp := ST(0);
        ST(0) := ST(1);
        ST(1) := temp;
FI;
```

FPU Flags Affected

C1	Set to 0.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FXRSTOR—Restore x87 FPU, MMX, XMM, and MXCSR State

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F AE /1 FXRSTOR m512byte	M	Valid	Valid	Restore the x87 FPU, MMX, XMM, and MXCSR register state from m512byte.
NP REX.W + 0F AE /1 FXRSTOR64 m512byte	M	Valid	N.E.	Restore the x87 FPU, MMX, XMM, and MXCSR register state from m512byte.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

Description

Reloads the x87 FPU, MMX technology, XMM, and MXCSR registers from the 512-byte memory image specified in the source operand. This data should have been written to memory previously using the FXSAVE instruction, and in the same format as required by the operating modes. The first byte of the data should be located on a 16-byte boundary. There are three distinct layouts of the FXSAVE state map: one for legacy and compatibility mode, a second format for 64-bit mode FXSAVE/FXRSTOR with REX.W=0, and the third format is for 64-bit mode with FXSAVE64/FXRSTOR64. Table 3-45 shows the layout of the legacy/compatibility mode state information in memory and describes the fields in the memory image for the FXRSTOR and FXSAVE instructions. Table 3-48 shows the layout of the 64-bit mode state information when REX.W is set (FXSAVE64/FXRSTOR64). Table 3-49 shows the layout of the 64-bit mode state information when REX.W is clear (FXSAVE/FXRSTOR).

The state image referenced with an FXRSTOR instruction must have been saved using an FXSAVE instruction or be in the same format as required by Table 3-45, Table 3-48, or Table 3-49. Referencing a state image saved with an FSAVE, FNSAVE instruction or incompatible field layout will result in an incorrect state restoration.

The FXRSTOR instruction does not flush pending x87 FPU exceptions. To check and raise exceptions when loading x87 FPU state information with the FXRSTOR instruction, use an FWAIT instruction after the FXRSTOR instruction.

If the OSFXSR bit in control register CR4 is not set, the FXRSTOR instruction may not restore the states of the XMM and MXCSR registers. This behavior is implementation dependent.

If the MXCSR state contains an unmasked exception with a corresponding status flag also set, loading the register with the FXRSTOR instruction will not result in a SIMD floating-point error condition being generated. Only the next occurrence of this unmasked exception will result in the exception being generated.

Bits 16 through 32 of the MXCSR register are defined as reserved and should be set to 0. Attempting to write a 1 in any of these bits from the saved state image will result in a general protection exception (#GP) being generated.

Bytes 464:511 of an FXSAVE image are available for software use. FXRSTOR ignores the content of bytes 464:511 in an FXSAVE state image.

Operation

```
IF 64-Bit Mode
    THEN
        (x87 FPU, MMX, XMM15-XMM0, MXCSR)    Load(SRC);
    ELSE
         (x87 FPU, MMX, XMM7-XMM0, MXCSR) := Load(SRC);
FI;
```

x87 FPU and SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment. (See alignment check exception [#AC] below.)
	For an attempt to set reserved bits in MXCSR.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
	If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[24] = 0.
	If instruction is preceded by a LOCK prefix.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
	For an attempt to set reserved bits in MXCSR.
#NM	If CR0.TS[bit 3] = 1.
	If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[24] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
	For an attempt to set reserved bits in MXCSR.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
	If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[24] = 0.
	If instruction is preceded by a LOCK prefix.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

FXSAVE—Save x87 FPU, MMX Technology, and SSE State

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F AE /0 FXSAVE m512byte	M	Valid	Valid	Save the x87 FPU, MMX, XMM, and MXCSR register state to m512byte.
NP REX.W + 0F AE /0 FXSAVE64 m512byte	M	Valid	N.E.	Save the x87 FPU, MMX, XMM, and MXCSR register state to m512byte.

Instruction Operand Encoding

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

Description

Saves the current state of the x87 FPU, MMX technology, XMM, and MXCSR registers to a 512-byte memory location specified in the destination operand. The content layout of the 512 byte region depends on whether the processor is operating in non-64-bit operating modes or 64-bit sub-mode of IA-32e mode.

Bytes 464:511 are available to software use. The processor does not write to bytes 464:511 of an FXSAVE area.

The operation of FXSAVE in non-64-bit modes is described first.

Non-64-Bit Mode Operation

Table 3-45 shows the layout of the state information in memory when the processor is operating in legacy modes.

Table 3-45. Non-64-Bit-Mode Layout of FXSAVE and FXRSTOR Memory Region

15 14	13 12	11 10	9 8	7 6	5	4	3 2	1 0	
Rsvd	FCS	FIP[31:0]		FOP	Rsvd	FTW	FSW	FCW	0
MXCSR_MASK		MXCSR		Rsrvd	FDS		FDP[31:0]		16
Reserved		ST0/MM0							32
Reserved		ST1/MM1							48
Reserved		ST2/MM2							64
Reserved		ST3/MM3							80
Reserved		ST4/MM4							96
Reserved		ST5/MM5							112
Reserved		ST6/MM6							128
Reserved		ST7/MM7							144
XMM0									160
XMM1									176
XMM2									192
XMM3									208
XMM4									224
XMM5									240
XMM6									256
XMM7									272
Reserved									288

Table 3-45. Non-64-Bit-Mode Layout of FXSAVE and FXRSTOR Memory Region (Contd.)

15 14	13 12	11 10	9 8	7 6	5	4	3 2	1 0	
Reserved									304
Reserved									320
Reserved									336
Reserved									352
Reserved									368
Reserved									384
Reserved									400
Reserved									416
Reserved									432
Reserved									448
Available									464
Available									480
Available									496

The destination operand contains the first byte of the memory image, and it must be aligned on a 16-byte boundary. A misaligned destination operand will result in a general-protection (#GP) exception being generated (or in some cases, an alignment check exception [#AC]).

The FXSAVE instruction is used when an operating system needs to perform a context switch or when an exception handler needs to save and examine the current state of the x87 FPU, MMX technology, and/or XMM and MXCSR registers.

The fields in Table 3-45 are defined in Table 3-46.

Table 3-46. Field Definitions

Field	Definition
FCW	x87 FPU Control Word (16 bits). See Figure 8-6 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for the layout of the x87 FPU control word.
FSW	x87 FPU Status Word (16 bits). See Figure 8-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for the layout of the x87 FPU status word.
Abridged FTW	x87 FPU Tag Word (8 bits). The tag information saved here is abridged, as described in the following paragraphs.
FOP	x87 FPU Opcode (16 bits). The lower 11 bits of this field contain the opcode, upper 5 bits are reserved. See Figure 8-8 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for the layout of the x87 FPU opcode field.
FIP	x87 FPU Instruction Pointer Offset (64 bits). The contents of this field differ depending on the current addressing mode (32-bit, 16-bit, or 64-bit) of the processor when the FXSAVE instruction was executed: 32-bit mode — 32-bit IP offset. 16-bit mode — low 16 bits are IP offset; high 16 bits are reserved. 64-bit mode with REX.W — 64-bit IP offset. 64-bit mode without REX.W — 32-bit IP offset. See "x87 FPU Instruction and Operand (Data) Pointers" in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for a description of the x87 FPU instruction pointer.
FCS	x87 FPU Instruction Pointer Selector (16 bits). If CPUID.07H.00H:EBX[13] = 1, the processor deprecates FCS and FDS, and this field is saved as 0000H.

Table 3-46. Field Definitions (Contd.)

Field	Definition
FDP	x87 FPU Instruction Operand (Data) Pointer Offset (64 bits). The contents of this field differ depending on the current addressing mode (32-bit, 16-bit, or 64-bit) of the processor when the FXSAVE instruction was executed: 32-bit mode — 32-bit DP offset. 16-bit mode — low 16 bits are DP offset; high 16 bits are reserved. 64-bit mode with REX.W — 64-bit DP offset. 64-bit mode without REX.W — 32-bit DP offset. See "x87 FPU Instruction and Operand (Data) Pointers" in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for a description of the x87 FPU operand pointer.
FDS	x87 FPU Instruction Operand (Data) Pointer Selector (16 bits). If CPUID.07H.00H:EBX[13] = 1, the processor deprecates FCS and FDS, and this field is saved as 0000H.
MXCSR	MXCSR Register State (32 bits). See Figure 10-3 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for the layout of the MXCSR register. If the OSFXSR bit in control register CR4 is not set, the FXSAVE instruction may not save this register. This behavior is implementation dependent.
MXCSR_ MASK	MXCSR_MASK (32 bits). This mask can be used to adjust values written to the MXCSR register, ensuring that reserved bits are set to 0. Set the mask bits and flags in MXCSR to the mode of operation desired for SSE and SSE2 SIMD floating-point instructions. See "Guidelines for Writing to the MXCSR Register" in Chapter 11 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for instructions for how to determine and use the MXCSR_MASK value.
ST0/MM0 through ST7/MM7	x87 FPU or MMX technology registers. These 80-bit fields contain the x87 FPU data registers or the MMX technology registers, depending on the state of the processor prior to the execution of the FXSAVE instruction. If the processor had been executing x87 FPU instruction prior to the FXSAVE instruction, the x87 FPU data registers are saved; if it had been executing MMX instructions (or SSE or SSE2 instructions that operated on the MMX technology registers), the MMX technology registers are saved. When the MMX technology registers are saved, the high 16 bits of the field are reserved.
XMM0 through XMM7	XMM registers (128 bits per field). If the OSFXSR bit in control register CR4 is not set, the FXSAVE instruction may not save these registers. This behavior is implementation dependent.

The FXSAVE instruction saves an abridged version of the x87 FPU tag word in the FTW field (unlike the FSAVE instruction, which saves the complete tag word). The tag information is saved in physical register order (R0 through R7), rather than in top-of-stack (TOS) order. With the FXSAVE instruction, however, only a single bit (1 for valid or 0 for empty) is saved for each tag. For example, assume that the tag word is currently set as follows:

R7	R6	R5	R4	R3	R2	R1	R0
11	xx	xx	xx	11	11	11	11

Here, 11B indicates empty stack elements and "xx" indicates valid (00B), zero (01B), or special (10B).

For this example, the FXSAVE instruction saves only the following 8 bits of information:

R7	R6	R5	R4	R3	R2	R1	R0
0	1	1	1	0	0	0	0

Here, a 1 is saved for any valid, zero, or special tag, and a 0 is saved for any empty tag.

The operation of the FXSAVE instruction differs from that of the FSAVE instruction, the as follows:

- FXSAVE instruction does not check for pending unmasked floating-point exceptions. (The FXSAVE operation in this regard is similar to the operation of the FNSAVE instruction).
- After the FXSAVE instruction has saved the state of the x87 FPU, MMX technology, XMM, and MXCSR registers, the processor retains the contents of the registers. Because of this behavior, the FXSAVE instruction cannot be used by an application program to pass a "clean" x87 FPU state to a procedure, since it retains the current state. To clean the x87 FPU state, an application must explicitly execute an FINIT instruction after an FXSAVE instruction to reinitialize the x87 FPU state.

- The format of the memory image saved with the FXSAVE instruction is the same regardless of the current addressing mode (32-bit or 16-bit) and operating mode (protected, real address, or system management). This behavior differs from the FSAVE instructions, where the memory image format is different depending on the addressing mode and operating mode. Because of the different image formats, the memory image saved with the FXSAVE instruction cannot be restored correctly with the FRSTOR instruction, and likewise the state saved with the FSAVE instruction cannot be restored correctly with the FXRSTOR instruction.

The FSAVE format for FTW can be recreated from the FTW valid bits and the stored 80-bit floating-point data (assuming the stored data was not the contents of MMX technology registers) using Table 3-47.

Table 3-47. Recreating FSAVE Format

Exponent all 1's	Exponent all 0's	Fraction all 0's	J and M bits	FTW valid bit	x87 FTW	
0	0	0	0x	1	Special	10
0	0	0	1x	1	Valid	00
0	0	1	00	1	Special	10
0	0	1	10	1	Valid	00
0	1	0	0x	1	Special	10
0	1	0	1x	1	Special	10
0	1	1	00	1	Zero	01
0	1	1	10	1	Special	10
1	0	0	1x	1	Special	10
1	0	0	1x	1	Special	10
1	0	1	00	1	Special	10
1	0	1	10	1	Special	10
For all legal combinations above.				0	Empty	11

The J-bit is defined to be the 1-bit binary integer to the left of the decimal place in the significand. The M-bit is defined to be the most significant bit of the fractional portion of the significand (i.e., the bit immediately to the right of the decimal place).

When the M-bit is the most significant bit of the fractional portion of the significand, it must be 0 if the fraction is all 0's.

IA-32e Mode Operation

In compatibility sub-mode of IA-32e mode, legacy SSE registers, XMM0 through XMM7, are saved according to the legacy FXSAVE map. In 64-bit mode, all of the SSE registers, XMM0 through XMM15, are saved. Additionally, there are two different layouts of the FXSAVE map in 64-bit mode, corresponding to FXSAVE64 (which requires REX.W=1) and FXSAVE (REX.W=0). In the FXSAVE64 map (Table 3-48), the FPU IP and FPU DP pointers are 64-bit wide. In the FXSAVE map for 64-bit mode (Table 3-49), the FPU IP and FPU DP pointers are 32-bits.

Table 3-48. Layout of the 64-Bit Mode FXSAVE64 Map (Requires REX.W = 1)

15 14	13 12	11 10	9 8	7 6	5	4	3 2	1 0	
FIP				FOP	Reserved	FTW	FSW	FCW	**0**
MXCSR_MASK		MXCSR		FDP					**16**
Reserved				ST0/MM0					**32**
Reserved				ST1/MM1					**48**
Reserved				ST2/MM2					**64**
Reserved				ST3/MM3					**80**
Reserved				ST4/MM4					**96**
Reserved				ST5/MM5					**112**
Reserved				ST6/MM6					**128**
Reserved				ST7/MM7					**144**
XMM0									**160**
XMM1									**176**
XMM2									**192**
XMM3									**208**
XMM4									**224**
XMM5									**240**
XMM6									**256**
XMM7									**272**
XMM8									**288**
XMM9									**304**
XMM10									**320**
XMM11									**336**
XMM12									**352**
XMM13									**368**
XMM14									**384**
XMM15									**400**
Reserved									**416**
Reserved									**432**
Reserved									**448**
Available									**464**
Available									**480**
Available									**496**

Table 3-49. Layout of the 64-Bit Mode FXSAVE Map (REX.W = 0)

15 14	13 12	11 10	9 8	7 6	5	4	3 2	1 0	
Reserved	FCS	FIP[31:0]		FOP	Reserved	FTW	FSW	FCW	0
MXCSR_MASK		MXCSR		Reserved	FDS		FDP[31:0]		16
Reserved		ST0/MM0							32
Reserved		ST1/MM1							48
Reserved		ST2/MM2							64
Reserved		ST3/MM3							80
Reserved		ST4/MM4							96
Reserved		ST5/MM5							112
Reserved		ST6/MM6							128
Reserved		ST7/MM7							144
XMM0									160
XMM1									176
XMM2									192
XMM3									208
XMM4									224
XMM5									240
XMM6									256
XMM7									272
XMM8									288
XMM9									304
XMM10									320
XMM11									336
XMM12									352
XMM13									368
XMM14									384
XMM15									400
Reserved									416
Reserved									432
Reserved									448
Available									464
Available									480
Available									496

Operation

```
IF 64-Bit Mode
    THEN
        IF REX.W = 1
            THEN
                DEST := Save64BitPromotedFxsave(x87 FPU, MMX, XMM15-XMM0,
                MXCSR);
            ELSE
                DEST := Save64BitDefaultFxsave(x87 FPU, MMX, XMM15-XMM0, MXCSR);
        FI;
    ELSE
        DEST := SaveLegacyFxsave(x87 FPU, MMX, XMM7-XMM0, MXCSR);
FI;
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment. (See the description of the alignment check exception [#AC] below.)
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
	If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[24] = 0.
#UD	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
	If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[24] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
	If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[24] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Implementation Note

The order in which the processor signals general-protection (#GP) and page-fault (#PF) exceptions when they both occur on an instruction boundary is given in Table 5-2 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B. This order vary for FXSAVE for different processor implementations.

FXTRACT—Extract Exponent and Significand

Opcode/ Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F4 FXTRACT	Valid	Valid	Separate value in ST(0) into exponent and significand, store exponent in ST(0), and push the significand onto the register stack.

Description

Separates the source value in the ST(0) register into its exponent and significand, stores the exponent in ST(0), and pushes the significand onto the register stack. Following this operation, the new top-of-stack register ST(0) contains the value of the original significand expressed as a floating-point value. The sign and significand of this value are the same as those found in the source operand, and the exponent is 3FFFH (biased value for a true exponent of zero). The ST(1) register contains the value of the original operand's true (unbiased) exponent expressed as a floating-point value. (The operation performed by this instruction is a superset of the IEEE-recommended logb(x) function.)

This instruction and the F2XM1 instruction are useful for performing power and range scaling operations. The FXTRACT instruction is also useful for converting numbers in double extended-precision floating-point format to decimal representations (e.g., for printing or displaying).

If the floating-point zero-divide exception (#Z) is masked and the source operand is zero, an exponent value of $-\infty$ is stored in register ST(1) and 0 with the sign of the source operand is stored in register ST(0).

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

Operation

```
TEMP := Significand(ST(0));
ST(0) := Exponent(ST(0));
TOP := TOP – 1;
ST(0) := TEMP;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; set to 1 if stack overflow occurred.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow or overflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#Z	ST(0) operand is ±0.
#D	Source operand is a denormal value.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FYL2X—Compute y ∗ $\log_2 x$

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F1	FYL2X	Valid	Valid	Replace ST(1) with (ST(1) ∗ $\log_2$ST(0)) and pop the register stack.

Description

Computes (ST(1) ∗ $\log_2$ (ST(0))), stores the result in register ST(1), and pops the FPU register stack. The source operand in ST(0) must be a non-zero positive number.

The following table shows the results obtained when taking the log of various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-50. FYL2X Results

	ST(0)								
ST(1)		− ∞	− F	±0	+0<+F<+1	+ 1	+ F > + 1	+ ∞	NaN
	− ∞	*	*	+ ∞	+ ∞	*	− ∞	− ∞	NaN
	− F	*	*	**	+ F	− 0	− F	− ∞	NaN
	− 0	*	*	*	+ 0	− 0	− 0	*	NaN
	+ 0	*	*	*	− 0	+ 0	+ 0	*	NaN
	+ F	*	*	**	− F	+ 0	+ F	+ ∞	NaN
	+ ∞	*	*	− ∞	− ∞	*	+ ∞	+ ∞	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-operation (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

If the divide-by-zero exception is masked and register ST(0) contains ±0, the instruction returns ∞ with a sign that is the opposite of the sign of the source operand in register ST(1).

The FYL2X instruction is designed with a built-in multiplication to optimize the calculation of logarithms with an arbitrary positive base (b):

$\log_b x := (\log_2 b)^{-1} * \log_2 x$

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

Operation

```
ST(1) := ST(1) ∗ log2ST(0);
PopRegisterStack;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Either operand is an SNaN or unsupported format. Source operand in register ST(0) is a negative finite value (not -0).
#Z	Source operand in register ST(0) is ±0.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FYL2XP1—Compute y ∗ $\log_2$(x +1)

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F9	FYL2XP1	Valid	Valid	Replace ST(1) with ST(1) ∗ $\log_2$(ST(0) + 1.0) and pop the register stack.

Description

Computes (ST(1) ∗ $\log_2$(ST(0) + 1.0)), stores the result in register ST(1), and pops the FPU register stack. The source operand in ST(0) must be in the range:

$$-(1-\sqrt{2}/2))\text{to}(1-\sqrt{2}/2)$$

The source operand in ST(1) can range from –∞ to +∞. If the ST(0) operand is outside of its acceptable range, the result is undefined and software should not rely on an exception being generated. Under some circumstances exceptions may be generated when ST(0) is out of range, but this behavior is implementation specific and not guaranteed.

The following table shows the results obtained when taking the log epsilon of various classes of numbers, assuming that underflow does not occur.

Table 3-51. FYL2XP1 Results

		ST(0)				
		–(1 – ($\sqrt{2}/2$)) to –0	-0	+0	+0 to +(1 - ($\sqrt{2}/2$))	NaN
ST(1)	– ∞	+∞	*	*	– ∞	NaN
	– F	+F	+0	-0	– F	NaN
	– 0	+0	+0	-0	– 0	NaN
	+0	– 0	– 0	+0	+0	NaN
	+F	– F	– 0	+0	+F	NaN
	+∞	– ∞	*	*	+∞	NaN
	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-operation (#IA) exception.

This instruction provides optimal accuracy for values of epsilon [the value in register ST(0)] that are close to 0. For small epsilon (ε) values, more significant digits can be retained by using the FYL2XP1 instruction than by using (ε+1) as an argument to the FYL2X instruction. The (ε+1) expression is commonly found in compound interest and annuity calculations. The result can be simply converted into a value in another logarithm base by including a scale factor in the ST(1) source operand. The following equation is used to calculate the scale factor for a particular logarithm base, where n is the logarithm base desired for the result of the FYL2XP1 instruction:

scale factor := $\log_n 2$

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

Operation

ST(1) := ST(1) ∗ $\log_2$(ST(0) + 1.0);
PopRegisterStack;

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Either operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

GF2P8AFFINEINVQB—Galois Field Affine Transformation Inverse

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F3A CF /r /ib GF2P8AFFINEINVQB xmm1, xmm2/m128, imm8	A	V/V	GFNI	Computes inverse affine transformation in the finite field GF(2^8).
VEX.128.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX GFNI	Computes inverse affine transformation in the finite field GF(2^8).
VEX.256.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX GFNI	Computes inverse affine transformation in the finite field GF(2^8).
EVEX.128.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	C	V/V	(AVX512VL OR AVX10.1) GFNI	Computes inverse affine transformation in the finite field GF(2^8).
EVEX.256.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	C	V/V	(AVX512VL OR AVX10.1) GFNI	Computes inverse affine transformation in the finite field GF(2^8).
EVEX.512.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	C	V/V	(AVX512F OR AVX10.1) GFNI	Computes inverse affine transformation in the finite field GF(2^8).

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8 (r)	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

The AFFINEINVB instruction computes an affine transformation in the Galois Field 2^8. For this instruction, an affine transformation is defined by A * inv(x) + b where "A" is an 8 by 8 bit matrix, and "x" and "b" are 8-bit vectors. The inverse of the bytes in x is defined with respect to the reduction polynomial $x^8 + x^4 + x^3 + x + 1$.

One SIMD register (operand 1) holds "x" as either 16, 32 or 64 8-bit vectors. A second SIMD (operand 2) register or memory operand contains 2, 4, or 8 "A" values, which are operated upon by the correspondingly aligned 8 "x" values in the first register. The "b" vector is constant for all calculations and contained in the immediate byte.

The EVEX encoded form of this instruction does not support memory fault suppression. The SSE encoded forms of the instruction require 16B alignment on their memory operations.

The inverse of each byte is given by the following table. The upper nibble is on the vertical axis and the lower nibble is on the horizontal axis. For example, the inverse of 0x95 is 0x8A.

Table 3-52. Inverse Byte Listings

-	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	8D	F6	CB	52	7B	D1	E8	4F	29	C0	B0	E1	E5	C7
1	74	B4	AA	4B	99	2B	60	5F	58	3F	FD	CC	FF	40	EE	B2
2	3A	6E	5A	F1	55	4D	A8	C9	C1	A	98	15	30	44	A2	C2
3	2C	45	92	6C	F3	39	66	42	F2	35	20	6F	77	BB	59	19
4	1D	FE	37	67	2D	31	F5	69	A7	64	AB	13	54	25	E9	9
5	ED	5C	5	CA	4C	24	87	BF	18	3E	22	F0	51	EC	61	17
6	16	5E	AF	D3	49	A6	36	43	F4	47	91	DF	33	93	21	3B
7	79	B7	97	85	10	B5	BA	3C	B6	70	D0	6	A1	FA	81	82
8	83	7E	7F	80	96	73	BE	56	9B	9E	95	D9	F7	2	B9	A4
9	DE	6A	32	6D	D8	8A	84	72	2A	14	9F	88	F9	DC	89	9A
A	FB	7C	2E	C3	8F	B8	65	48	26	C8	12	4A	CE	E7	D2	62
B	C	E0	1F	EF	11	75	78	71	A5	8E	76	3D	BD	BC	86	57
C	B	28	2F	A3	DA	D4	E4	F	A9	27	53	4	1B	FC	AC	E6
D	7A	7	AE	63	C5	DB	E2	EA	94	8B	C4	D5	9D	F8	90	6B
E	B1	D	D6	EB	C6	E	CF	AD	8	4E	D7	E3	5D	50	1E	B3
F	5B	23	38	34	68	46	3	8C	DD	9C	7D	A0	CD	1A	41	1C

Operation

```
define affine_inverse_byte(tsrc2qw, src1byte, imm):
    FOR i := 0 to 7:
        * parity(x) = 1 if x has an odd number of 1s in it, and 0 otherwise.*
        * inverse(x) is defined in the table above *
        retbyte.bit[i] := parity(tsrc2qw.byte[7-i] AND inverse(src1byte)) XOR imm8.bit[i]
    return retbyte
```

VGF2P8AFFINEINVQB dest, src1, src2, imm8 (EVEX Encoded Version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1:
    IF SRC2 is memory and EVEX.b==1:
        tsrc2 := SRC2.qword[0]
    ELSE:
        tsrc2 := SRC2.qword[j]

FOR b := 0 to 7:
    IF k1[j*8+b] OR *no writemask*:
        FOR i := 0 to 7:
            DEST.qword[j].byte[b] := affine_inverse_byte(tsrc2, SRC1.qword[j].byte[b], imm8)
    ELSE IF *zeroing*:
        DEST.qword[j].byte[b] := 0
    *ELSE DEST.qword[j].byte[b] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VGF2P8AFFINEINVQB dest, src1, src2, imm8 (128b and 256b VEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256)
FOR j := 0 TO KL-1:
    FOR b := 0 to 7:
        DEST.qword[j].byte[b] := affine_inverse_byte(SRC2.qword[j], SRC1.qword[j].byte[b], imm8)
DEST[MAX_VL-1:VL] := 0
```

GF2P8AFFINEINVQB srcdest, src1, imm8 (128b SSE Encoded Version)

```
FOR j := 0 TO 1:
    FOR b := 0 to 7:
        SRCDEST.qword[j].byte[b] := affine_inverse_byte(SRC1.qword[j], SRCDEST.qword[j].byte[b], imm8)
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)GF2P8AFFINEINVQB __m128i _mm_gf2p8affineinv_epi64_epi8(__m128i, __m128i, int);
(V)GF2P8AFFINEINVQB __m128i _mm_mask_gf2p8affineinv_epi64_epi8(__m128i, __mmask16, __m128i, __m128i, int);
(V)GF2P8AFFINEINVQB __m128i _mm_maskz_gf2p8affineinv_epi64_epi8(__mmask16, __m128i, __m128i, int);
VGF2P8AFFINEINVQB __m256i _mm256_gf2p8affineinv_epi64_epi8(__m256i, __m256i, int);
VGF2P8AFFINEINVQB __m256i _mm256_mask_gf2p8affineinv_epi64_epi8(__m256i, __mmask32, __m256i, __m256i, int);
VGF2P8AFFINEINVQB __m256i _mm256_maskz_gf2p8affineinv_epi64_epi8(__mmask32, __m256i, __m256i, int);
VGF2P8AFFINEINVQB __m512i _mm512_gf2p8affineinv_epi64_epi8(__m512i, __m512i, int);
VGF2P8AFFINEINVQB __m512i _mm512_mask_gf2p8affineinv_epi64_epi8(__m512i, __mmask64, __m512i, __m512i, int);
VGF2P8AFFINEINVQB __m512i _mm512_maskz_gf2p8affineinv_epi64_epi8(__mmask64, __m512i, __m512i, int);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Legacy-encoded and VEX-encoded: See Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded: See Table 2-52, “Type E4NF Class Exception Conditions.”

GF2P8AFFINEQB—Galois Field Affine Transformation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F3A CE /r /ib GF2P8AFFINEQB xmm1, xmm2/m128, imm8	A	V/V	GFNI	Computes affine transformation in the finite field GF(2^8).
VEX.128.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX GFNI	Computes affine transformation in the finite field GF(2^8).
VEX.256.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX GFNI	Computes affine transformation in the finite field GF(2^8).
EVEX.128.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	C	V/V	(AVX512VL OR AVX10.1) GFNI	Computes affine transformation in the finite field GF(2^8).
EVEX.256.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	C	V/V	(AVX512VL OR AVX10.1) GFNI	Computes affine transformation in the finite field GF(2^8).
EVEX.512.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	C	V/V	(AVX512F OR AVX10.1) GFNI	Computes affine transformation in the finite field GF(2^8).

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8 (r)	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

The AFFINEB instruction computes an affine transformation in the Galois Field 2^8. For this instruction, an affine transformation is defined by A * x + b where "A" is an 8 by 8 bit matrix, and "x" and "b" are 8-bit vectors. One SIMD register (operand 1) holds "x" as either 16, 32 or 64 8-bit vectors. A second SIMD (operand 2) register or memory operand contains 2, 4, or 8 "A" values, which are operated upon by the correspondingly aligned 8 "x" values in the first register. The "b" vector is constant for all calculations and contained in the immediate byte.

The EVEX encoded form of this instruction does not support memory fault suppression. The SSE encoded forms of the instruction require16B alignment on their memory operations.

Operation

```
define parity(x):
    t := 0                  // single bit
    FOR i := 0 to 7:
        t = t xor x.bit[i]
    return t

define affine_byte(tsrc2qw, src1byte, imm):
    FOR i := 0 to 7:
        * parity(x) = 1 if x has an odd number of 1s in it, and 0 otherwise.*
        retbyte.bit[i] := parity(tsrc2qw.byte[7-i] AND src1byte) XOR imm8.bit[i]
    return retbyte
```

VGF2P8AFFINEQB dest, src1, src2, imm8 (EVEX Encoded Version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1:
    IF SRC2 is memory and EVEX.b==1:
        tsrc2 := SRC2.qword[0]
    ELSE:
        tsrc2 := SRC2.qword[j]

    FOR b := 0 to 7:
        IF k1[j*8+b] OR *no writemask*:
            DEST.qword[j].byte[b] := affine_byte(tsrc2, SRC1.qword[j].byte[b], imm8)
        ELSE IF *zeroing*:
            DEST.qword[j].byte[b] := 0
        *ELSE DEST.qword[j].byte[b] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VGF2P8AFFINEQB dest, src1, src2, imm8 (128b and 256b VEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256)
FOR j := 0 TO KL-1:
    FOR b := 0 to 7:
        DEST.qword[j].byte[b] := affine_byte(SRC2.qword[j], SRC1.qword[j].byte[b], imm8)
DEST[MAX_VL-1:VL] := 0
```

GF2P8AFFINEQB srcdest, src1, imm8 (128b SSE Encoded Version)

```
FOR j := 0 TO 1:
    FOR b := 0 to 7:
        SRCDEST.qword[j].byte[b] := affine_byte(SRC1.qword[j], SRCDEST.qword[j].byte[b], imm8)
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)GF2P8AFFINEQB __m128i _mm_gf2p8affine_epi64_epi8(__m128i, __m128i, int);
(V)GF2P8AFFINEQB __m128i _mm_mask_gf2p8affine_epi64_epi8(__m128i, __mmask16, __m128i, __m128i, int);
(V)GF2P8AFFINEQB __m128i _mm_maskz_gf2p8affine_epi64_epi8(__mmask16, __m128i, __m128i, int);
VGF2P8AFFINEQB __m256i _mm256_gf2p8affine_epi64_epi8(__m256i, __m256i, int);
VGF2P8AFFINEQB __m256i _mm256_mask_gf2p8affine_epi64_epi8(__m256i, __mmask32, __m256i, __m256i, int);
VGF2P8AFFINEQB __m256i _mm256_maskz_gf2p8affine_epi64_epi8(__mmask32, __m256i, __m256i, int);
VGF2P8AFFINEQB __m512i _mm512_gf2p8affine_epi64_epi8(__m512i, __m512i, int);
VGF2P8AFFINEQB __m512i _mm512_mask_gf2p8affine_epi64_epi8(__m512i, __mmask64, __m512i, __m512i, int);
VGF2P8AFFINEQB __m512i _mm512_maskz_gf2p8affine_epi64_epi8(__mmask64, __m512i, __m512i, int);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Legacy-encoded and VEX-encoded: See Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded: See Table 2-52, "Type E4NF Class Exception Conditions."

GF2P8MULB—Galois Field Multiply Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F38 CF /r GF2P8MULB xmm1, xmm2/m128	A	V/V	GFNI	Multiplies elements in the finite field GF(2^8).
VEX.128.66.0F38.W0 CF /r VGF2P8MULB xmm1, xmm2, xmm3/m128	B	V/V	AVX GFNI	Multiplies elements in the finite field GF(2^8).
VEX.256.66.0F38.W0 CF /r VGF2P8MULB ymm1, ymm2, ymm3/m256	B	V/V	AVX GFNI	Multiplies elements in the finite field GF(2^8).
EVEX.128.66.0F38.W0 CF /r VGF2P8MULB xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL OR AVX10.1) GFNI	Multiplies elements in the finite field GF(2^8).
EVEX.256.66.0F38.W0 CF /r VGF2P8MULB ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL OR AVX10.1) GFNI	Multiplies elements in the finite field GF(2^8).
EVEX.512.66.0F38.W0 CF /r VGF2P8MULB zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	(AVX512F OR AVX10.1) GFNI	Multiplies elements in the finite field GF(2^8).

Instruction Operand Encoding

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

Description

The instruction multiplies elements in the finite field GF(2^8), operating on a byte (field element) in the first source operand and the corresponding byte in a second source operand. The field GF(2^8) is represented in polynomial representation with the reduction polynomial $x^8 + x^4 + x^3 + x + 1$.

This instruction does not support broadcasting.

The EVEX encoded form of this instruction supports memory fault suppression. The SSE encoded forms of the instruction require16B alignment on their memory operations.

Operation

```
define gf2p8mul_byte(src1byte, src2byte):
    tword := 0
    FOR i := 0 to 7:
        IF src2byte.bit[i]:
            tword := tword XOR (src1byte<< i)
        * carry out polynomial reduction by the characteristic polynomial p*
    FOR i := 14 downto 8:
        p := 0x11B << (i-8)          *0x11B = 0000_0001_0001_1011 in binary*
        IF tword.bit[i]:
            tword := tword XOR p
return tword.byte[0]
```

VGF2P8MULB dest, src1, src2 (EVEX Encoded Version)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.byte[j] := gf2p8mul_byte(SRC1.byte[j], SRC2.byte[j])
    ELSE IF *zeroing*:
        DEST.byte[j] := 0
    * ELSE DEST.byte[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VGF2P8MULB dest, src1, src2 (128b and 256b VEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256)
FOR j := 0 TO KL-1:
    DEST.byte[j] := gf2p8mul_byte(SRC1.byte[j], SRC2.byte[j])
DEST[MAX_VL-1:VL] := 0
```

GF2P8MULB srcdest, src1 (128b SSE Encoded Version)

```
FOR j := 0 TO 15:
    SRCDEST.byte[j] :=gf2p8mul_byte(SRCDEST.byte[j], SRC1.byte[j])
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)GF2P8MULB __m128i _mm_gf2p8mul_epi8(__m128i, __m128i);
(V)GF2P8MULB __m128i _mm_mask_gf2p8mul_epi8(__m128i, __mmask16, __m128i, __m128i);
(V)GF2P8MULB __m128i _mm_maskz_gf2p8mul_epi8(__mmask16, __m128i, __m128i);
VGF2P8MULB __m256i _mm256_gf2p8mul_epi8(__m256i, __m256i);
VGF2P8MULB __m256i _mm256_mask_gf2p8mul_epi8(__m256i, __mmask32, __m256i, __m256i);
VGF2P8MULB __m256i _mm256_maskz_gf2p8mul_epi8(__mmask32, __m256i, __m256i);
VGF2P8MULB __m512i _mm512_gf2p8mul_epi8(__m512i, __m512i);
VGF2P8MULB __m512i _mm512_mask_gf2p8mul_epi8(__m512i, __mmask64, __m512i, __m512i);
VGF2P8MULB __m512i _mm512_maskz_gf2p8mul_epi8(__mmask64, __m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Legacy-encoded and VEX-encoded: See Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded: See Table 2-51, “Type E4 Class Exception Conditions.”

HADDPD—Packed Double Precision Floating-Point Horizontal Add

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 7C /r HADDPD xmm1, xmm2/m128	RM	V/V	SSE3	Horizontal add packed double precision floating-point values from xmm2/m128 to xmm1.
VEX.128.66.0F.WIG 7C /r VHADDPD xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Horizontal add packed double precision floating-point values from xmm2 and xmm3/mem.
VEX.256.66.0F.WIG 7C /r VHADDPD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Horizontal add packed double precision floating-point values from ymm2 and ymm3/mem.

Instruction Operand Encoding

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

Description

Adds the double precision floating-point values in the high and low quadwords of the destination operand and stores the result in the low quadword of the destination operand.

Adds the double precision floating-point values in the high and low quadwords of the source operand and stores the result in the high quadword of the destination operand.

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

See Figure 3-12 for HADDPD; see Figure 3-13 for VHADDPD.

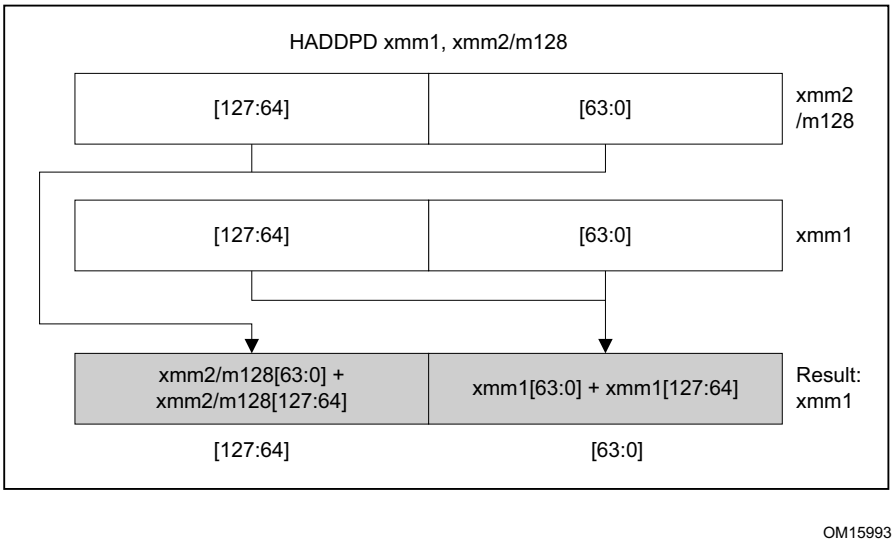


Figure 3-12. HADDPD—Packed Double Precision Floating-Point Horizontal Add

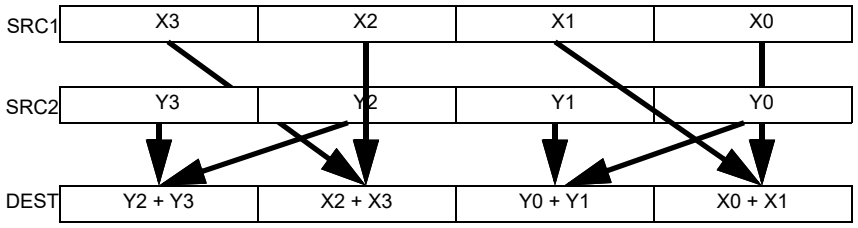


Figure 3-13. VHADDPD Operation

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

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

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

Operation

HADDPD (128-bit Legacy SSE Version)

```
DEST[63:0] := SRC1[127:64] + SRC1[63:0]
DEST[127:64] := SRC2[127:64] + SRC2[63:0]
DEST[MAXVL-1:128] (Unmodified)
```

VHADDPD (VEX.128 Encoded Version)

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

VHADDPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[127:64] + SRC1[63:0]
DEST[127:64] := SRC2[127:64] + SRC2[63:0]
DEST[191:128] := SRC1[255:192] + SRC1[191:128]
DEST[255:192] := SRC2[255:192] + SRC2[191:128]
```

Intel C/C++ Compiler Intrinsic Equivalent

VHADDPD __m256d _mm256_hadd_pd (__m256d a, __m256d b);
HADDPD __m128d _mm_hadd_pd (__m128d a, __m128d b);

Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Table 2-19, "Type 2 Class Exception Conditions."

HADDPS—Packed Single Precision Floating-Point Horizontal Add

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 7C /r HADDPS xmm1, xmm2/m128	RM	V/V	SSE3	Horizontal add packed single precision floating-point values from xmm2/m128 to xmm1.
VEX.128.F2.0F.WIG 7C /r VHADDPS xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Horizontal add packed single precision floating-point values from xmm2 and xmm3/mem.
VEX.256.F2.0F.WIG 7C /r VHADDPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Horizontal add packed single precision floating-point values from ymm2 and ymm3/mem.

Instruction Operand Encoding

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

Description

Adds the single precision floating-point values in the first and second dwords of the destination operand and stores the result in the first dword of the destination operand.

Adds single precision floating-point values in the third and fourth dword of the destination operand and stores the result in the second dword of the destination operand.

Adds single precision floating-point values in the first and second dword of the source operand and stores the result in the third dword of the destination operand.

Adds single precision floating-point values in the third and fourth dword of the source operand and stores the result in the fourth dword of the destination operand.

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

See Figure 3-14 for HADDPS; see Figure 3-15 for VHADDPS.

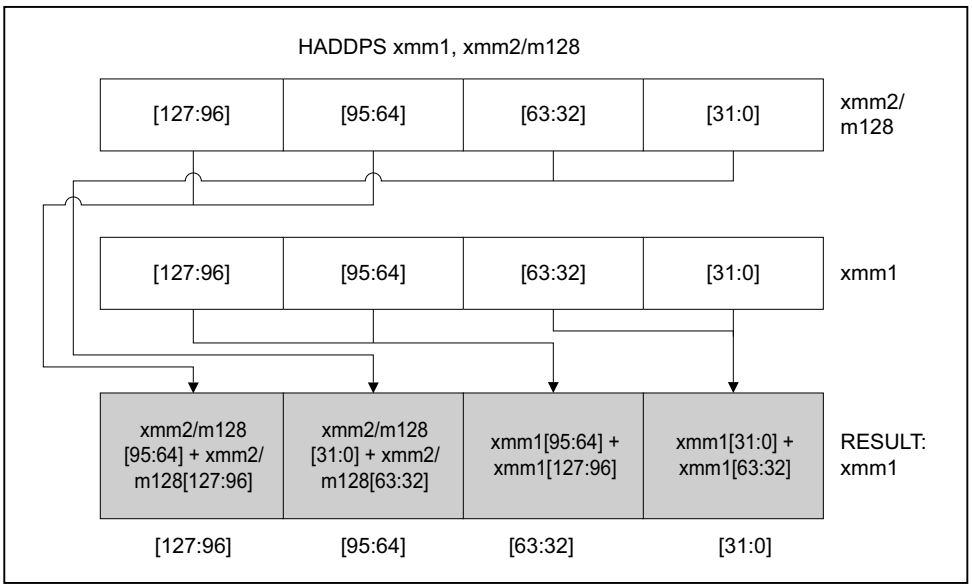


Figure 3-14. HADDPS—Packed Single Precision Floating-Point Horizontal Add

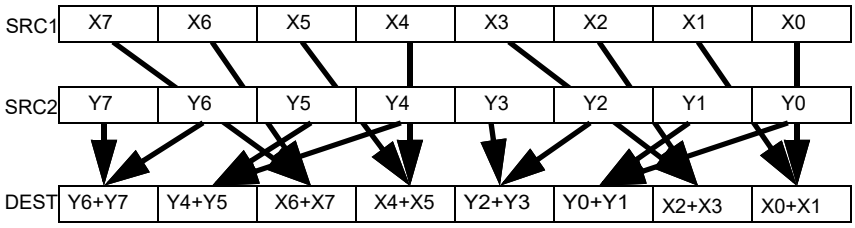


Figure 3-15. VHADDPS Operation

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

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

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

Operation

HADDPS (128-bit Legacy SSE Version)

```
DEST[31:0] := SRC1[63:32] + SRC1[31:0]
DEST[63:32] := SRC1[127:96] + SRC1[95:64]
DEST[95:64] := SRC2[63:32] + SRC2[31:0]
DEST[127:96] := SRC2[127:96] + SRC2[95:64]
DEST[MAXVL-1:128] (Unmodified)
```

VHADDPS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[63:32] + SRC1[31:0]
DEST[63:32] := SRC1[127:96] + SRC1[95:64]
DEST[95:64] := SRC2[63:32] + SRC2[31:0]
DEST[127:96] := SRC2[127:96] + SRC2[95:64]
DEST[MAXVL-1:128] := 0
```

VHADDPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[63:32] + SRC1[31:0]
DEST[63:32] := SRC1[127:96] + SRC1[95:64]
DEST[95:64] := SRC2[63:32] + SRC2[31:0]
DEST[127:96] := SRC2[127:96] + SRC2[95:64]
DEST[159:128] := SRC1[191:160] + SRC1[159:128]
DEST[191:160] := SRC1[255:224] + SRC1[223:192]
DEST[223:192] := SRC2[191:160] + SRC2[159:128]
DEST[255:224] := SRC2[255:224] + SRC2[223:192]
```

Intel C/C++ Compiler Intrinsic Equivalent

HADDPS __m128 _mm_hadd_ps (__m128 a, __m128 b);
VHADDPS __m256 _mm256_hadd_ps (__m256 a, __m256 b);

Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Table 2-19, "Type 2 Class Exception Conditions."

HLT—Halt

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F4	HLT	ZO	Valid	Valid	Halt

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Stops instruction execution and places the processor in a HALT state. An enabled interrupt (including NMI and SMI), a debug exception, the BINIT# signal, the INIT# signal, or the RESET# signal will resume execution. If an interrupt (including NMI) is used to resume execution after a HLT instruction, the saved instruction pointer (CS:EIP) points to the instruction following the HLT instruction.

When a HLT instruction is executed on an Intel 64 or IA-32 processor supporting Intel Hyper-Threading Technology, only the logical processor that executes the instruction is halted. The other logical processors in the physical processor remain active, unless they are each individually halted by executing a HLT instruction.

The HLT instruction is a privileged instruction. When the processor is running in protected or virtual-8086 mode, the privilege level of a program or procedure must be 0 to execute the HLT instruction.

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

Operation

```
Enter Halt state;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

HRESET—History Reset

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 3A F0 C0 /ib HRESET imm8, <EAX>	A	V/V	HRESET	Processor history reset request. Controlled by the EAX implicit operand.

Instruction Operand Encoding

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

Description

Requests the processor to selectively reset selected components of hardware history maintained by the current logical processor. HRESET operation is controlled by the implicit EAX operand. The value of the explicit imm8 operand is ignored. This instruction can only be executed at privilege level 0.

The HRESET instruction can be used to request reset of multiple components of hardware history. Prior to the execution of HRESET, the system software must take the following steps:

1. Enumerate the HRESET capabilities via CPUID.20H.00H:EBX, which indicates what components of hardware history can be reset.
2. Only the bits enumerated by CPUID.20H.00H:EBX can be set in the IA32_HRESET_ENABLE MSR.

HRESET causes a general-protection exception (#GP) if EAX sets any bits that are not set in the IA32_HRESET_ENABLE MSR.

Any attempt to execute the HRESET instruction inside a transactional region will result in a transaction abort.

Operation

```
IF EAX = 0
  THEN NOP
  ELSE
    FOREACH i such that EAX[i] = 1
      Reset prediction history for feature i
FI
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If CPL > 0 or (EAX AND NOT IA32_HRESET_ENABLE) ≠0.
#UD	If CPUID.07H.01H:EAX.HRESET[22] = 0.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

#GP(0)	HRESET instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

HSUBPD—Packed Double Precision Floating-Point Horizontal Subtract

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 7D /r HSUBPD xmm1, xmm2/m128	RM	V/V	SSE3	Horizontal subtract packed double precision floating-point values from xmm2/m128 to xmm1.
VEX.128.66.0F.WIG 7D /r VHSUBPD xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Horizontal subtract packed double precision floating-point values from xmm2 and xmm3/mem.
VEX.256.66.0F.WIG 7D /r VHSUBPD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Horizontal subtract packed double precision floating-point values from ymm2 and ymm3/mem.

Instruction Operand Encoding

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

Description

The HSUBPD instruction subtracts horizontally the packed double precision floating-point numbers of both operands.

Subtracts the double precision floating-point value in the high quadword of the destination operand from the low quadword of the destination operand and stores the result in the low quadword of the destination operand.

Subtracts the double precision floating-point value in the high quadword of the source operand from the low quadword of the source operand and stores the result in the high quadword of the destination operand.

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

See Figure 3-16 for HSUBPD; see Figure 3-17 for VHSUBPD.

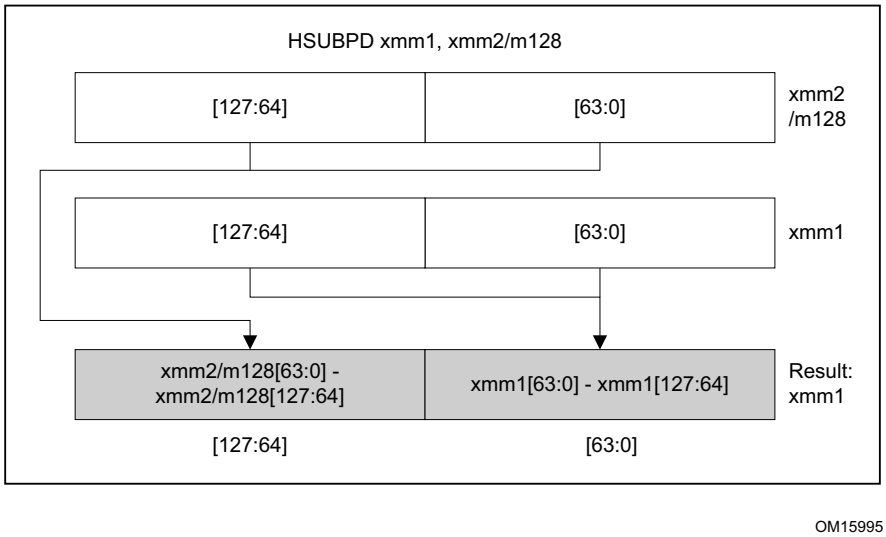


Figure 3-16. HSUBPD—Packed Double Precision Floating-Point Horizontal Subtract

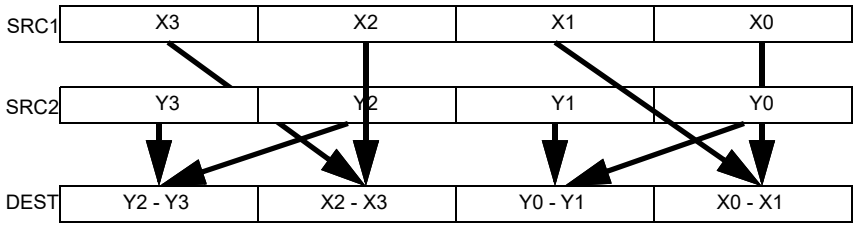


Figure 3-17. VHSUBPD operation

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

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

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

Operation

HSUBPD (128-bit Legacy SSE Version)

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

VHSUBPD (VEX.128 Encoded Version)

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

VHSUBPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0] - SRC1[127:64]
DEST[127:64] := SRC2[63:0] - SRC2[127:64]
DEST[191:128] := SRC1[191:128] - SRC1[255:192]
DEST[255:192] := SRC2[191:128] - SRC2[255:192]
```

Intel C/C++ Compiler Intrinsic Equivalent

HSUBPD __m128d _mm_hsub_pd(__m128d a, __m128d b)
VHSUBPD __m256d _mm256_hsub_pd (__m256d a, __m256d b);

Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Table 2-19, "Type 2 Class Exception Conditions."

HSUBPS—Packed Single Precision Floating-Point Horizontal Subtract

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 7D /r HSUBPS xmm1, xmm2/m128	RM	V/V	SSE3	Horizontal subtract packed single precision floating-point values from xmm2/m128 to xmm1.
VEX.128.F2.0F.WIG 7D /r VHSUBPS xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Horizontal subtract packed single precision floating-point values from xmm2 and xmm3/mem.
VEX.256.F2.0F.WIG 7D /r VHSUBPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Horizontal subtract packed single precision floating-point values from ymm2 and ymm3/mem.

Instruction Operand Encoding

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

Description

Subtracts the single precision floating-point value in the second dword of the destination operand from the first dword of the destination operand and stores the result in the first dword of the destination operand.

Subtracts the single precision floating-point value in the fourth dword of the destination operand from the third dword of the destination operand and stores the result in the second dword of the destination operand.

Subtracts the single precision floating-point value in the second dword of the source operand from the first dword of the source operand and stores the result in the third dword of the destination operand.

Subtracts the single precision floating-point value in the fourth dword of the source operand from the third dword of the source operand and stores the result in the fourth dword of the destination operand.

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

See Figure 3-18 for HSUBPS; see Figure 3-19 for VHSUBPS.

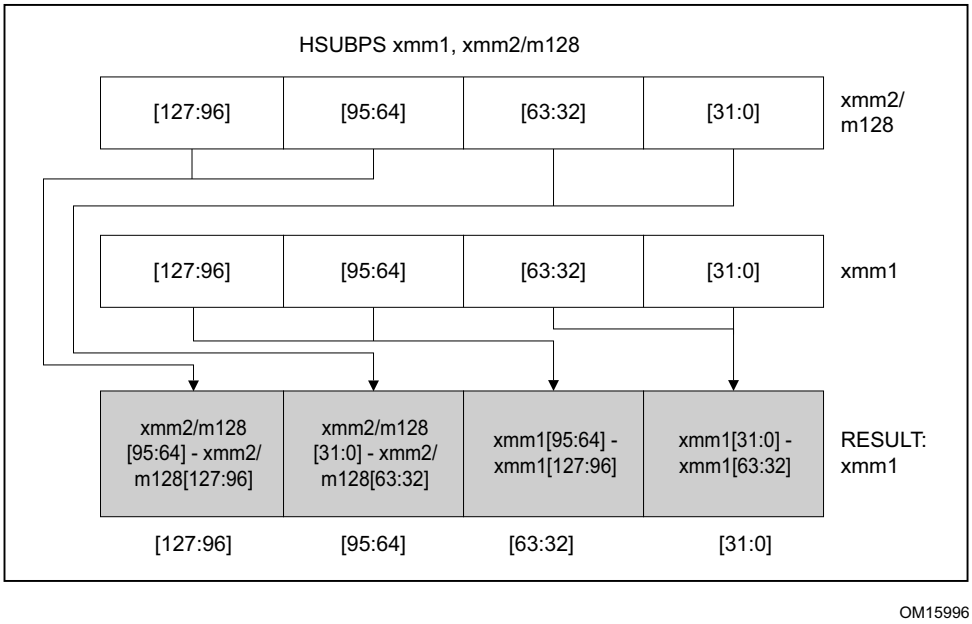


Figure 3-18. HSUBPS—Packed Single Precision Floating-Point Horizontal Subtract

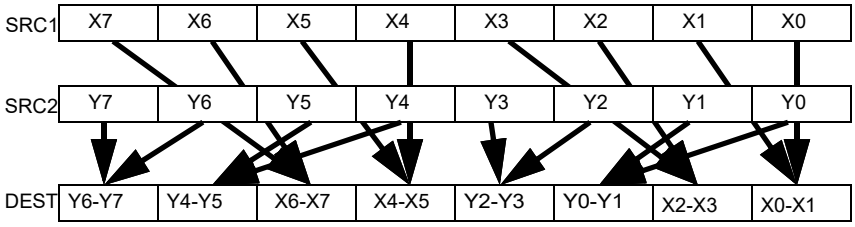


Figure 3-19. VHSUBPS Operation

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

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

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

Operation

HSUBPS (128-bit Legacy SSE Version)

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

VHSUBPS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] - SRC1[63:32]
DEST[63:32] := SRC1[95:64] - SRC1[127:96]
DEST[95:64] := SRC2[31:0] - SRC2[63:32]
DEST[127:96] := SRC2[95:64] - SRC2[127:96]
DEST[MAXVL-1:128] := 0
```

VHSUBPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[31:0] - SRC1[63:32]
DEST[63:32] := SRC1[95:64] - SRC1[127:96]
DEST[95:64] := SRC2[31:0] - SRC2[63:32]
DEST[127:96] := SRC2[95:64] - SRC2[127:96]
DEST[159:128] := SRC1[159:128] - SRC1[191:160]
DEST[191:160] := SRC1[223:192] - SRC1[255:224]
DEST[223:192] := SRC2[159:128] - SRC2[191:160]
DEST[255:224] := SRC2[223:192] - SRC2[255:224]
```

Intel C/C++ Compiler Intrinsic Equivalent

HSUBPS __m128 _mm_hsub_ps(__m128 a, __m128 b);
VHSUBPS __m256 _mm256_hsub_ps (__m256 a, __m256 b);

Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Table 2-19, "Type 2 Class Exception Conditions."

IDIV—Signed Divide

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /7	IDIV r/m8[1]	M	Valid	Valid	Signed divide AX by r/m8, with result stored in: AL := Quotient, AH := Remainder.
F7 /7	IDIV r/m16	M	Valid	Valid	Signed divide DX:AX by r/m16, with result stored in AX := Quotient, DX := Remainder.
F7 /7	IDIV r/m32	M	Valid	Valid	Signed divide EDX:EAX by r/m32, with result stored in EAX := Quotient, EDX := Remainder.
REX.W + F7 /7	IDIV r/m64	M	Valid	N.E.	Signed divide RDX:RAX by r/m64, with result stored in RAX := Quotient, RDX := Remainder.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

Description

Divides the (signed) value in the AX, DX:AX, or EDX:EAX (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, or EDX:EAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size (dividend/divisor).

Non-integral results are truncated (chopped) towards 0. The remainder is always less than the divisor in magnitude. Overflow is indicated with the #DE (divide error) exception rather than with the CF flag.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. In 64-bit mode when REX.W is applied, the instruction divides the signed value in RDX:RAX by the source operand. RAX contains a 64-bit quotient; RDX contains a 64-bit remainder.

See the summary chart at the beginning of this section for encoding data and limits. See Table 3-53.

Table 3-53. IDIV Results

Operand Size	Dividend	Divisor	Quotient	Remainder	Quotient Range
Word/byte	AX	r/m8	AL	AH	–128 to +127
Doubleword/word	DX:AX	r/m16	AX	DX	–32,768 to +32,767
Quadword/doubleword	EDX:EAX	r/m32	EAX	EDX	-2^{31} to $2^{31} - 1$
Doublequadword/ quadword	RDX:RAX	r/m64	RAX	RDX	-2^{63} to $2^{63} - 1$

Operation

```
IF SRC = 0
    THEN #DE; (* Divide error *)
FI;

IF OperandSize = 8 (* Word/byte operation *)
    THEN
        temp := AX / SRC; (* Signed division *)
        IF (temp > 7FH) or (temp < 80H)
        (* If a positive result is greater than 7FH or a negative result is less than 80H *)
            THEN #DE; (* Divide error *)
            ELSE
                AL := temp;
                AH := AX SignedModulus SRC;
        FI;
    ELSE IF OperandSize = 16 (* Doubleword/word operation *)
        THEN
            temp := DX:AX / SRC; (* Signed division *)
            IF (temp > 7FFFH) or (temp < 8000H)
            (* If a positive result is greater than 7FFFH
            or a negative result is less than 8000H *)
                THEN
                    #DE; (* Divide error *)
                ELSE
                    AX := temp;
                    DX := DX:AX SignedModulus SRC;
            FI;
        FI;
    ELSE IF OperandSize = 32 (* Quadword/doubleword operation *)
        THEN
            temp := EDX:EAX / SRC; (* Signed division *)
            IF (temp > 7FFFFFFFH) or (temp < 80000000H)
            (* If a positive result is greater than 7FFFFFFFH
            or a negative result is less than 80000000H *)
                THEN
                    #DE; (* Divide error *)
                ELSE
                    EAX := temp;
                    EDX := EDX:EAX SignedModulus SRC;
            FI;
        FI;
    ELSE IF OperandSize = 64 (* Doublequadword/quadword operation *)
        THEN
            temp := RDX:RAX / SRC; (* Signed division *)
            IF (temp > 7FFFFFFFFFFFFFFFH) or (temp < 8000000000000000H)
            (* If a positive result is greater than 7FFFFFFFFFFFFFFFH
            or a negative result is less than 8000000000000000H *)
                THEN
                    #DE; (* Divide error *)
                ELSE
                    RAX := temp;
                    RDX := RDX:RAX SignedModulus SRC;
            FI;
        FI;
```

FI;

Flags Affected

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

Protected Mode Exceptions

#DE	If the source operand (divisor) is 0.
	The signed result (quotient) is too large for the destination.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#DE	If the source operand (divisor) is 0.
	The signed result (quotient) is too large for the destination.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

IMUL—Signed Multiply

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /5	IMUL r/m8[1]	M	Valid	Valid	AX:= AL ∗ r/m byte.
F7 /5	IMUL r/m16	M	Valid	Valid	DX:AX := AX ∗ r/m word.
F7 /5	IMUL r/m32	M	Valid	Valid	EDX:EAX := EAX ∗ r/m32.
REX.W + F7 /5	IMUL r/m64	M	Valid	N.E.	RDX:RAX := RAX ∗ r/m64.
0F AF /r	IMUL r16, r/m16	RM	Valid	Valid	Word register := word register ∗ r/m16.
0F AF /r	IMUL r32, r/m32	RM	Valid	Valid	Doubleword register := doubleword register ∗ r/m32.
REX.W + 0F AF /r	IMUL r64, r/m64	RM	Valid	N.E.	Quadword register := Quadword register ∗ r/m64.
6B /r ib	IMUL r16, r/m16, imm8	RMI	Valid	Valid	Word register := r/m16 ∗ sign-extended immediate byte.
6B /r ib	IMUL r32, r/m32, imm8	RMI	Valid	Valid	Doubleword register := r/m32 ∗ sign-extended immediate byte.
REX.W + 6B /r ib	IMUL r64, r/m64, imm8	RMI	Valid	N.E.	Quadword register := r/m64 ∗ sign-extended immediate byte.
69 /r iw	IMUL r16, r/m16, imm16	RMI	Valid	Valid	Word register := r/m16 ∗ immediate word.
69 /r id	IMUL r32, r/m32, imm32	RMI	Valid	Valid	Doubleword register := r/m32 ∗ immediate doubleword.
REX.W + 69 /r id	IMUL r64, r/m64, imm32	RMI	Valid	N.E.	Quadword register := r/m64 ∗ immediate doubleword.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	N/A	N/A	N/A
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8/16/32	N/A

Description

Performs a signed multiplication of two operands. This instruction has three forms, depending on the number of operands.

- **One-operand form** — This form is identical to that used by the MUL instruction. Here, the source operand (in a general-purpose register or memory location) is multiplied by the value in the AL, AX, EAX, or RAX register (depending on the operand size) and the product (twice the size of the input operand) is stored in the AX, DX:AX, EDX:EAX, or RDX:RAX registers, respectively.
- **Two-operand form** — With this form the destination operand (the first operand) is multiplied by the source operand (second operand). The destination operand is a general-purpose register and the source operand is an immediate value, a general-purpose register, or a memory location. The intermediate product (twice the size of the input operand) is truncated and stored in the destination operand location.
- **Three-operand form** — This form requires a destination operand (the first operand) and two source operands (the second and the third operands). Here, the first source operand (which can be a general-purpose register or a memory location) is multiplied by the second source operand (an immediate value). The intermediate product (twice the size of the first source operand) is truncated and stored in the destination operand (a general-purpose register).

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The CF and OF flags are set when the signed integer value of the intermediate product differs from the sign extended operand-size-truncated product, otherwise the CF and OF flags are cleared.

The three forms of the IMUL instruction are similar in that the length of the product is calculated to twice the length of the operands. With the one-operand form, the product is stored exactly in the destination. With the two- and three- operand forms, however, the result is truncated to the length of the destination before it is stored in the destination register. Because of this truncation, the CF or OF flag should be tested to ensure that no significant bits are lost.

The two- and three-operand forms may also be used with unsigned operands because the lower half of the product is the same regardless if the operands are signed or unsigned. The CF and OF flags, however, cannot be used to determine if the upper half of the result is non-zero.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. Use of REX.W modifies the three forms of the instruction as follows.

- **One-operand form** —The source operand (in a 64-bit general-purpose register or memory location) is multiplied by the value in the RAX register and the product is stored in the RDX:RAX registers.
- **Two-operand form** — The source operand is promoted to 64 bits if it is a register or a memory location. The destination operand is promoted to 64 bits.
- **Three-operand form** — The first source operand (either a register or a memory location) and destination operand are promoted to 64 bits. If the source operand is an immediate, it is sign extended to 64 bits.

Operation

```
IF (NumberOfOperands = 1)
    THEN IF (OperandSize = 8)
        THEN
            TMP_XP := AL * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *);
            AX := TMP_XP[15:0];
            IF SignExtend(TMP_XP[7:0]) = TMP_XP
                THEN CF := 0; OF := 0;
                ELSE CF := 1; OF := 1; FI;
        ELSE IF OperandSize = 16
            THEN
                TMP_XP := AX * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
                DX:AX := TMP_XP[31:0];
                IF SignExtend(TMP_XP[15:0]) = TMP_XP
                    THEN CF := 0; OF := 0;
                    ELSE CF := 1; OF := 1; FI;
            ELSE IF OperandSize = 32
                THEN
                    TMP_XP := EAX * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC*)
                    EDX:EAX := TMP_XP[63:0];
                    IF SignExtend(TMP_XP[31:0]) = TMP_XP
                        THEN CF := 0; OF := 0;
                        ELSE CF := 1; OF := 1; FI;
                ELSE (* OperandSize = 64 *)
                    TMP_XP := RAX * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
                    RDX:RAX := TMP_XP[127:0];
                    IF SignExtend(TMP_XP[63:0]) = TMP_XP
                        THEN CF := 0; OF := 0;
                        ELSE CF := 1; OF := 1; FI;
                FI;
        FI;
```

```
    ELSE IF (NumberOfOperands = 2)
        THEN
            TMP_XP := DEST * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
            DEST := TruncateToOperandSize(TMP_XP);
            IF SignExtend(DEST) ≠ TMP_XP
                THEN CF := 1; OF := 1;
                ELSE CF := 0; OF := 0; FI;
        ELSE (* NumberOfOperands = 3 *)
            TMP_XP := SRC1 * SRC2 (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC1 *)
            DEST := TruncateToOperandSize(TMP_XP);
            IF SignExtend(DEST) ≠ TMP_XP
                THEN CF := 1; OF := 1;
                ELSE CF := 0; OF := 0; FI;
    FI;
FI;
```

Flags Affected

For the one operand form of the instruction, the CF and OF flags are set when significant bits are carried into the upper half of the result and cleared when the result fits exactly in the lower half of the result. For the two- and three-operand forms of the instruction, the CF and OF flags are set when the result must be truncated to fit in the destination operand size and cleared when the result fits exactly in the destination operand size. The SF, ZF, AF, and PF flags are undefined.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

IN—Input From Port

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
E4 ib	IN AL, imm8	I	Valid	Valid	Input byte from imm8 I/O port address into AL.
E5 ib	IN AX, imm8	I	Valid	Valid	Input word from imm8 I/O port address into AX.
E5 ib	IN EAX, imm8	I	Valid	Valid	Input dword from imm8 I/O port address into EAX.
EC	IN AL,DX	ZO	Valid	Valid	Input byte from I/O port in DX into AL.
ED	IN AX,DX	ZO	Valid	Valid	Input word from I/O port in DX into AX.
ED	IN EAX,DX	ZO	Valid	Valid	Input doubleword from I/O port in DX into EAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	imm8	N/A	N/A	N/A
ZO	N/A	N/A	N/A	N/A

Description

Copies the value from the I/O port specified with the second operand (source operand) to the destination operand (first operand). The source operand can be a byte-immediate or the DX register; the destination operand can be register AL, AX, or EAX, depending on the size of the port being accessed (8, 16, or 32 bits, respectively). Using the DX register as a source operand allows I/O port addresses from 0 to 65,535 to be accessed; using a byte immediate allows I/O port addresses 0 to 255 to be accessed.

When accessing an 8-bit I/O port, the opcode determines the port size; when accessing a 16- and 32-bit I/O port, the operand-size attribute determines the port size. At the machine code level, I/O instructions are shorter when accessing 8-bit I/O ports. Here, the upper eight bits of the port address will be 0.

This instruction is only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 20, "Input/Output," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for more information on accessing I/O ports in the I/O address space.

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

Operation

```
IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
    THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
        IF (Any I/O Permission Bit for I/O port being accessed = 1)
            THEN (* I/O operation is not allowed *)
                #GP(0);
            ELSE ( * I/O operation is allowed *)
                DEST := SRC; (* Read from selected I/O port *)
        FI;
    ELSE (Real Mode or Protected Mode with CPL ≤ IOPL *)
        DEST := SRC; (* Read from selected I/O port *)
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

INC—Increment by 1

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FE /0	INC r/m8[1]	M	Valid	Valid	Increment r/m byte by 1.
FF /0	INC r/m16	M	Valid	Valid	Increment r/m word by 1.
FF /0	INC r/m32	M	Valid	Valid	Increment r/m doubleword by 1.
REX.W + FF /0	INC r/m64	M	Valid	N.E.	Increment r/m quadword by 1.
40+ rw[2]	INC r16	O	N.E.	Valid	Increment word register by 1.
40+ rd	INC r32	O	N.E.	Valid	Increment doubleword register by 1.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.
2. 40H through 47H are REX prefixes in 64-bit mode.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	N/A	N/A	N/A
O	opcode + rd (r, w)	N/A	N/A	N/A

Description

Adds 1 to the destination operand, while preserving the state of the CF flag. The destination operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (Use a ADD instruction with an immediate operand of 1 to perform an increment operation that does updates the CF flag.)

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, INC r16 and INC r32 are not encodable (because opcodes 40H through 47H are REX prefixes). Otherwise, the instruction's 64-bit mode default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

Operation

```
DEST := DEST + 1;
```

Flags Affected

The CF flag is not affected. The OF, SF, ZF, AF, and PF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination operand is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULLsegment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

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

#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

INCSSPD/INCSSPQ—Increment Shadow Stack Pointer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F AE /5 INCSSPD r32	R	V/V	CET_SS	Increment SSP by 4 * r32[7:0].
F3 REX.W 0F AE /5 INCSSPQ r64	R	V/N.E.	CET_SS	Increment SSP by 8 * r64[7:0].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
R	N/A	ModRM:r/m (r)	N/A	N/A	N/A

Description

This instruction can be used to increment the current shadow stack pointer by the operand size of the instruction times the unsigned 8-bit value specified by bits 7:0 in the source operand. The instruction performs a pop and discard of the first and last element on the shadow stack in the range specified by the unsigned 8-bit value in bits 7:0 of the source operand.

Operation

```
IF CPL = 3
    IF (CR4.CET & IA32_U_CET.SH_STK_EN) = 0
        THEN #UD; FI;
ELSE
    IF (CR4.CET & IA32_S_CET.SH_STK_EN) = 0
        THEN #UD; FI;
FI;

IF (operand size is 64-bit)
    THEN
        Range := R64[7:0];
        shadow_stack_load 8 bytes from SSP;
        IF Range > 0
            THEN shadow_stack_load 8 bytes from SSP + 8 * (Range - 1);
        FI;
        SSP := SSP + Range * 8;
    ELSE
        Range := R32[7:0];
        shadow_stack_load 4 bytes from SSP;
        IF Range > 0
            THEN shadow_stack_load 4 bytes from SSP + 4 * (Range - 1);
        FI;
        SSP := SSP + Range * 4;
FI;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

INCSSPD void _incsspd(int);
INCSSPQ void _incsspq(int);

Protected Mode Exceptions

#UD	If the LOCK prefix is used. If CR4.CET = 0. IF CPL = 3 and IA32_U_CET.SH_STK_EN = 0. IF CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
#PF(fault-code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	The INCSSP instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The INCSSP instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

INSERTPS—Insert Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 21 /r ib INSERTPS xmm1, xmm2/m32, imm8	A	V/V	SSE4_1	Insert a single precision floating-point value selected by imm8 from xmm2/m32 into xmm1 at the specified destination element specified by imm8 and zero out destination elements in xmm1 as indicated in imm8.
VEX.128.66.0F3A.WIG 21 /r ib VINSERTPS xmm1, xmm2, xmm3/m32, imm8	B	V/V	AVX	Insert a single precision floating-point value selected by imm8 from xmm3/m32 and merge with values in xmm2 at the specified destination element specified by imm8 and write out the result and zero out destination elements in xmm1 as indicated in imm8.
EVEX.128.66.0F3A.W0 21 /r ib VINSERTPS xmm1, xmm2, xmm3/m32, imm8	C	V/V	AVX512F OR AVX10.1	Insert a single precision floating-point value selected by imm8 from xmm3/m32 and merge with values in xmm2 at the specified destination element specified by imm8 and write out the result and zero out destination elements in xmm1 as indicated in imm8.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

(register source form)

Copy a single precision scalar floating-point element into a 128-bit vector register. The immediate operand has three fields, where the ZMask bits specify which elements of the destination will be set to zero, the Count_D bits specify which element of the destination will be overwritten with the scalar value, and for vector register sources the Count_S bits specify which element of the source will be copied. When the scalar source is a memory operand the Count_S bits are ignored.

(memory source form)

Load a floating-point element from a 32-bit memory location and destination operand it into the first source at the location indicated by the Count_D bits of the immediate operand. Store in the destination and zero out destination elements based on the ZMask bits of the immediate operand.

128-bit Legacy SSE version: The first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

VEX.128 and EVEX encoded version: The destination and first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

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

Operation

VINSERTPS (VEX.128 and EVEX Encoded Version)

```
IF (SRC = REG) THEN COUNT_S := imm8[7:6]
    ELSE COUNT_S := 0
COUNT_D := imm8[5:4]
ZMASK := imm8[3:0]
CASE (COUNT_S) OF
    0: TMP := SRC2[31:0]
    1: TMP := SRC2[63:32]
    2: TMP := SRC2[95:64]
    3: TMP := SRC2[127:96]
ESAC;
CASE (COUNT_D) OF
    0: TMP2[31:0] := TMP
        TMP2[127:32] := SRC1[127:32]
    1: TMP2[63:32] := TMP
        TMP2[31:0] := SRC1[31:0]
        TMP2[127:64] := SRC1[127:64]
    2: TMP2[95:64] := TMP
        TMP2[63:0] := SRC1[63:0]
        TMP2[127:96] := SRC1[127:96]
    3: TMP2[127:96] := TMP
        TMP2[95:0] := SRC1[95:0]
ESAC;

IF (ZMASK[0] = 1) THEN DEST[31:0] := 00000000H
    ELSE DEST[31:0] := TMP2[31:0]
IF (ZMASK[1] = 1) THEN DEST[63:32] := 00000000H
    ELSE DEST[63:32] := TMP2[63:32]
IF (ZMASK[2] = 1) THEN DEST[95:64] := 00000000H
    ELSE DEST[95:64] := TMP2[95:64]
IF (ZMASK[3] = 1) THEN DEST[127:96] := 00000000H
    ELSE DEST[127:96] := TMP2[127:96]
DEST[MAXVL-1:128] := 0
```

INSERTPS (128-bit Legacy SSE Version)

```
IF (SRC = REG) THEN COUNT_S :=imm8[7:6]
    ELSE COUNT_S :=0
COUNT_D := imm8[5:4]
ZMASK := imm8[3:0]
CASE (COUNT_S) OF
    0: TMP := SRC[31:0]
    1: TMP := SRC[63:32]
    2: TMP := SRC[95:64]
    3: TMP := SRC[127:96]
ESAC;

CASE (COUNT_D) OF
    0: TMP2[31:0] := TMP
        TMP2[127:32] := DEST[127:32]
    1: TMP2[63:32] := TMP
        TMP2[31:0] := DEST[31:0]
        TMP2[127:64] := DEST[127:64]
    2: TMP2[95:64] := TMP
```

```
        TMP2[63:0] := DEST[63:0]
        TMP2[127:96] := DEST[127:96]
    3: TMP2[127:96] := TMP
        TMP2[95:0] := DEST[95:0]
ESAC;

IF (ZMASK[0] = 1) THEN DEST[31:0] := 00000000H
    ELSE DEST[31:0] := TMP2[31:0]
IF (ZMASK[1] = 1) THEN DEST[63:32] := 00000000H
    ELSE DEST[63:32] := TMP2[63:32]
IF (ZMASK[2] = 1) THEN DEST[95:64] := 00000000H
    ELSE DEST[95:64] := TMP2[95:64]
IF (ZMASK[3] = 1) THEN DEST[127:96] := 00000000H
    ELSE DEST[127:96] := TMP2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VINSERTPS __m128 _mm_insert_ps(__m128 dst, __m128 src, const int nidx);
INSETRTPS __m128 _mm_insert_ps(__m128 dst, __m128 src, const int nidx);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, "Type 5 Class Exception Conditions," additionally:

#UD	If VEX.L = 0.

EVEX-encoded instruction, see Table 2-59, "Type E9NF Class Exception Conditions."

INS/INSB/INSW/INSD—Input from Port to String

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
6C	INS m8, DX	ZO	Valid	Valid	Input byte from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.[1]
6D	INS m16, DX	ZO	Valid	Valid	Input word from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.[1]
6D	INS m32, DX	ZO	Valid	Valid	Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.[1]
6C	INSB	ZO	Valid	Valid	Input byte from I/O port specified in DX into memory location specified with ES:(E)DI or RDI.[1]
6D	INSW	ZO	Valid	Valid	Input word from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.[1]
6D	INSD	ZO	Valid	Valid	Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.[1]

NOTES:

1. In 64-bit mode, only 64-bit (RDI) and 32-bit (EDI) address sizes are supported. In non-64-bit mode, only 32-bit (EDI) and 16-bit (DI) address sizes are supported.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Copies the data from the I/O port specified with the source operand (second operand) to the destination operand (first operand). The source operand is an I/O port address (from 0 to 65,535) that is read from the DX register. The destination operand is a memory location, the address of which is read from either the ES:DI, ES:EDI or the RDI registers (depending on the address-size attribute of the instruction, 16, 32 or 64, respectively). (The ES segment cannot be overridden with a segment override prefix.) The size of the I/O port being accessed (that is, the size of the source and destination operands) is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operands" form and the "no-operands" form. The explicit-operands form (specified with the INS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source operand must be "DX," and the destination operand should be a symbol that indicates the size of the I/O port and the destination address. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the destination operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the ES:(E)DI registers, which must be loaded correctly before the INS instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the INS instructions. Here also DX is assumed by the processor to be the source operand and ES:(E)DI is assumed to be the destination operand. The size of the I/O port is specified with the choice of mnemonic: INSB (byte), INSW (word), or INSD (doubleword).

After the byte, word, or doubleword is transfer from the I/O port to the memory location, the DI/EDI/RDI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)DI register is incremented; if the DF flag is 1, the (E)DI register is decremented.) The (E)DI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The INS, INSB, INSW, and INSD instructions can be preceded by the REP prefix for block input of ECX bytes, words, or doublewords. See "REP—Repeat String Operation (Prefix)" in Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B, for a description of the REP prefix.

These instructions are only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 20, "Input/Output," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for more information on accessing I/O ports in the I/O address space.

In 64-bit mode, default address size is 64 bits, 32 bit address size is supported using the prefix 67H. The address of the memory destination is specified by RDI or EDI. 16-bit address size is not supported in 64-bit mode. The operand size is not promoted.

These instructions may read from the I/O port without writing to the memory location if an exception or VM exit occurs due to the write (e.g. #PF). If this would be problematic, for example because the I/O port read has side-effects, software should ensure the write to the memory location does not cause an exception or VM exit.

Operation

```
IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
    THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
        IF (Any I/O Permission Bit for I/O port being accessed = 1)
            THEN (* I/O operation is not allowed *)
                #GP(0);
            ELSE (* I/O operation is allowed *)
                DEST := SRC; (* Read from I/O port *)
        FI;
    ELSE (Real Mode or Protected Mode with CPL IOPL *)
        DEST := SRC; (* Read from I/O port *)
FI;
```

Non-64-bit Mode:

```
IF (Byte transfer)
    THEN IF DF = 0
        THEN (E)DI := (E)DI + 1;
        ELSE (E)DI := (E)DI – 1; FI;
    ELSE IF (Word transfer)
        THEN IF DF = 0
            THEN (E)DI := (E)DI + 2;
            ELSE (E)DI := (E)DI – 2; FI;
        ELSE (* Doubleword transfer *)
            THEN IF DF = 0
                THEN (E)DI := (E)DI + 4;
                ELSE (E)DI := (E)DI – 4; FI;
        FI;
FI;
```

FI64-bit Mode:

```
IF (Byte transfer)
    THEN IF DF = 0
        THEN (E|R)DI := (E|R)DI + 1;
        ELSE (E|R)DI := (E|R)DI – 1; FI;
    ELSE IF (Word transfer)
        THEN IF DF = 0
            THEN (E)DI := (E)DI + 2;
            ELSE (E)DI := (E)DI – 2; FI;
        ELSE (* Doubleword transfer *)
            THEN IF DF = 0
                THEN (E|R)DI := (E|R)DI + 4;
```

```
                ELSE (E|R)DI := (E|R)DI – 4; FI;
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
	If the destination is located in a non-writable segment.
	If an illegal memory operand effective address in the ES segments is given.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

#GP(0)	If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

INT *n*/INTO/INT3/INT1—Call to Interrupt Procedure

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
CC	INT3	ZO	Valid	Valid	Generate breakpoint trap.
CD ib	INT imm8	I	Valid	Valid	Generate software interrupt with vector specified by immediate byte.
CE	INTO	ZO	Invalid	Valid	Generate overflow trap if overflow flag is 1.
F1	INT1	ZO	Valid	Valid	Generate debug trap.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A
I	imm8	N/A	N/A	N/A

Description

The INT *n* instruction generates a call to the interrupt or exception handler specified with the destination operand (see the section titled "Interrupts and Exceptions" in Chapter 6 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1). The destination operand specifies a vector from 0 to 255, encoded as an 8-bit unsigned intermediate value. Each vector provides an index to a gate descriptor in the IDT. The first 32 vectors are reserved by Intel for system use. Some of these vectors are used for internally generated exceptions.

When IDT event delivery is used, the vector provides an index to a gate descriptor in the IDT. The first 32 vectors are reserved by Intel for system use. Some of these vectors are used for internally generated exceptions. When FRED event delivery is used, the vector is saved on the stack of the event handler.

The INT *n* instruction is the general mnemonic for executing a software-generated call to an interrupt handler. The INTO instruction is a special mnemonic for calling overflow exception (#OF), exception 4. The overflow interrupt checks the OF flag in the EFLAGS register and calls the overflow interrupt handler if the OF flag is set to 1. (The INTO instruction cannot be used in 64-bit mode.)

The INT3 instruction uses a one-byte opcode (CC) and is intended for calling the debug exception handler with a breakpoint exception (#BP). (This one-byte form is useful because it can replace the first byte of any instruction at which a breakpoint is desired, including other one-byte instructions, without overwriting other instructions.)

The INT1 instruction also uses a one-byte opcode (F1) and generates a debug exception (#DB) without setting any bits in DR6.[1] Hardware vendors may use the INT1 instruction for hardware debug. For that reason, Intel recommends software vendors instead use the INT3 instruction for software breakpoints.

An interrupt generated by the INTO, INT3, or INT1 instruction differs from one generated by INT *n* in the following ways:

- The normal IOPL checks do not occur in virtual-8086 mode. The interrupt is taken (without fault) with any IOPL value.
- The interrupt redirection enabled by the virtual-8086 mode extensions (VME) does not occur. The interrupt is always handled by a protected-mode handler.
- FRED event delivery uses event type 4 (software interrupt) for INT *n* but uses event type 5 (privileged software exception) for INT1 and event type 6 (software exception) for INT3 and INTO.

(These features do not pertain to CD01, CD03, or CD04, the "normal" 2-byte opcodes for INT 1, INT 3, and INT 4, respectively. Intel and Microsoft assemblers will not generate the CD03 opcode from any mnemonic, but this opcode can be created by direct numeric code definition or by self-modifying code.)

The operation and use of these instructions depends significantly on whether FRED transitions have been enabled by setting CR4.FRED. If CR4.FRED = 0, IDT event delivery is used; otherwise, FRED event delivery is used. The following sections apply as indicated.

1. The mnemonic ICEBP has also been used for the instruction with opcode F1.

With IDT Event Delivery

With IDT event delivery, the action of the INT *n* instruction (including the INTO, INT3, and INT1 instructions) is similar to that of a far call made with the CALL instruction. The primary difference is that with the INT *n* instruction, the EFLAGS register is pushed onto the stack before the return address. (The return address is a far address consisting of the current values of the CS and EIP registers.) Returns from interrupt procedures are handled with the IRET instruction, which pops the EFLAGS information and return address from the stack.

Each of the INT *n*, INTO, and INT3 instructions generates a general-protection exception (#GP) if the CPL is greater than the DPL value in the selected gate descriptor in the IDT. In contrast, the INT1 instruction can deliver a #DB even if the CPL is greater than the DPL of descriptor 1 in the IDT. (This behavior supports the use of INT1 by hardware vendors performing hardware debug.)

The vector specifies an interrupt descriptor in the interrupt descriptor table (IDT); that is, it provides index into the IDT. The selected interrupt descriptor in turn contains a pointer to an interrupt or exception handler procedure. In protected mode, the IDT contains an array of 8-byte descriptors, each of which is an interrupt gate, trap gate, or task gate. In real-address mode, the IDT is an array of 4-byte far pointers (2-byte code segment selector and a 2-byte instruction pointer), each of which point directly to a procedure in the selected segment. (Note that in real-address mode, the IDT is called the **interrupt vector table**, and its pointers are called interrupt vectors.)

The following decision table indicates which action in the lower portion of the table is taken given the conditions in the upper portion of the table. Each Y in the lower section of the decision table represents a procedure defined in the "Operation" section for this instruction (except #GP).

Table 3-54. Decision Table

PE	0	1	1	1	1	1	1	1
VM	–	–	–	–	–	0	1	1
IOPL	–	–	–	–	–	–	<3	=3
DPL/CPL RELATIONSHIP	–	DPL< CPL	–	DPL> CPL	DPL= CPL or C	DPL< CPL & NC	–	–
INTERRUPT TYPE	–	S/W	–	–	–	–	–	–
GATE TYPE	–	–	Task	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt
REAL-ADDRESS-MODE	Y							
PROTECTED-MODE		Y	Y	Y	Y	Y	Y	Y
TRAP-OR-INTERRUPT-GATE				Y	Y	Y	Y	Y
INTER-PRIVILEGE-LEVEL-INTERRUPT						Y		
INTRA-PRIVILEGE-LEVEL-INTERRUPT					Y			
INTERRUPT-FROM-VIRTUAL-8086-MODE								Y
TASK-GATE			Y					
#GP		Y		Y			Y	

NOTES:

– Don't Care.

Y Yes, action taken.

Blank Action not taken.

S/W Applies to INT *n*, INT3, and INTO, but not to INT1.

When the processor is executing in virtual-8086 mode, the IOPL determines the action of the INT n instruction. If the IOPL is less than 3, the processor generates a #GP(selector) exception; if the IOPL is 3, the processor executes

a protected mode interrupt to privilege level 0. The interrupt gate's DPL must be set to 3 and the target CPL of the interrupt handler procedure must be 0 to execute the protected mode interrupt to privilege level 0.

The interrupt descriptor table register (IDTR) specifies the base linear address and limit of the IDT. The initial base address value of the IDTR after the processor is powered up or reset is 0.

Refer to Chapter 6, "Procedure Calls, Interrupts, and Exceptions" and Chapter 18, "Control-flow Enforcement Technology (CET)" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for CET details.

With FRED Event Delivery

When FRED transitions are enabled, all of these are events are delivered using FRED event delivery. Chapter 8, "Flexible Return and Event Delivery (FRED)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 provides details. Like IDT event delivery, FRED event delivery is similar to a far call made with the CALL instruction, but the FRED stack frame is different. Returns from FRED event delivery use either the ERETS instruction or the ERETU instruction, depending on the CPL at the time of the original event.

FRED transitions can be enabled only in IA-32e mode. They cannot be enabled in real-address mode, protected mode, or virtual-8086 mode. FRED event delivery does not use the IDT, and its operation thus does not depend on any descriptor or DPL value or the value of IOPL.

Instruction ordering. Instructions following an INT *n* may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the INT n have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible). This applies also to the INTO, INT3, and INT1 instructions, but not to executions of INTO when EFLAGS.OF = 0.

Operation

The following operational description applies not only to the INT *n,* INTO, INT3, or INT1 instructions, but also to the delivery of external interrupts, nonmaskable interrupts (NMIs), and exceptions. Some of these events push onto the stack an error code.

The operational description specifies numerous checks whose failure may result in delivery of a nested exception. In these cases, the original event is not delivered.

The operational description specifies the error code delivered by any nested exception. In some cases, the error code is specified with a pseudofunction error_code(num,idt,ext), where idt and ext are bit values. The pseudofunction produces an error code as follows: (1) if idt is 0, the error code is (num & FCH) | ext; (2) if idt is 1, the error code is (num « 3) | 2 | ext.

In many cases, the pseudofunction error_code is invoked with a pseudovariable EXT. The value of EXT depends on the nature of the event whose delivery encountered a nested exception: if that event is a software interrupt (INT *n*, INT3, or INTO), EXT is 0; otherwise (including INT1), EXT is 1.

```
IF PE = 0
    THEN
        GOTO REAL-ADDRESS-MODE;
    ELSE (* PE = 1 *)
        IF (EFLAGS.VM = 1 AND CR4.VME = 0 AND IOPL < 3 AND INT n)
            THEN
                #GP(0); (* Bit 0 of error code is 0 because INT n *)
            ELSE
                IF (EFLAGS.VM = 1 AND CR4.VME = 1 AND INT n)
                    THEN
                        Consult bit n of the software interrupt redirection bit map in the TSS;
                        IF bit n is clear
                            THEN (* redirect interrupt to 8086 program interrupt handler *)
                                Push EFLAGS[15:0];      (* if IOPL < 3, save VIF in IF position and save IOPL position as 3 *)
                                Push CS;
                                Push IP;
                                IF IOPL = 3
```

```
                                    THEN IF := 0; (* Clear interrupt flag *)
                                    ELSE VIF := 0; (* Clear virtual interrupt flag *)
                                FI;
                                TF := 0; (* Clear trap flag *)
                                load CS and EIP (lower 16 bits only) from entry n in interrupt vector table referenced from TSS;
                            ELSE
                                IF IOPL = 3
                                    THEN GOTO PROTECTED-MODE;
                                    ELSE #GP(0); (* Bit 0 of error code is 0 because INT n *)
                                FI;
                        FI;
                    ELSE (* Protected mode, IA-32e mode, or virtual-8086 mode interrupt *)
                        IF (IA32_EFER.LMA = 0)
                            THEN (* Protected mode, or virtual-8086 mode interrupt *)
                                GOTO PROTECTED-MODE;
                            ELSE (* IA-32e mode interrupt *)
                            GOTO IA-32e-MODE;
                        FI;
                FI;
        FI;
FI;
REAL-ADDRESS-MODE:
    IF ((vector_number « 2) + 3) is not within IDT limit
        THEN #GP; FI;
    IF stack not large enough for a 6-byte return information
        THEN #SS; FI;
    Push (EFLAGS[15:0]);
    IF := 0; (* Clear interrupt flag *)
    TF := 0; (* Clear trap flag *)
    AC := 0; (* Clear AC flag *)
    Push(CS);
    Push(IP);
    (* No error codes are pushed in real-address mode*)
    CS := IDT(Descriptor (vector_number « 2), selector));
    EIP := IDT(Descriptor (vector_number « 2), offset)); (* 16 bit offset AND 0000FFFFH *)
END;

PROTECTED-MODE:
    IF ((vector_number « 3) + 7) is not within IDT limits
    or selected IDT descriptor is not an interrupt-, trap-, or task-gate type
        THEN #GP(error_code(vector_number,1,EXT)); FI;
        (* idt operand to error_code set because vector is used *)
    IF software interrupt (* Generated by INT n, INT3, or INTO; does not apply to INT1 *)
        THEN
            IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
                THEN #GP(error_code(vector_number,1,0)); FI;
                (* idt operand to error_code set because vector is used *)
                (* ext operand to error_code is 0 because INT n, INT3, or INTO*)
    FI;
    IF gate not present
        THEN #NP(error_code(vector_number,1,EXT)); FI;
        (* idt operand to error_code set because vector is used *)
    IF task gate (* Specified in the selected interrupt table descriptor *)
        THEN GOTO TASK-GATE;
```

```
        ELSE GOTO TRAP-OR-INTERRUPT-GATE; (* PE = 1, trap/interrupt gate *)
    FI;
END;

IA-32e-MODE:
    IF INTO and CS.L = 1 (64-bit mode)
        THEN #UD;
    FI;
    IF CR4.FRED = 0
        THEN
            IF ((vector_number « 4) + 15) is not in IDT limits
            or selected IDT descriptor is not an interrupt-, or trap-gate type
                THEN #GP(error_code(vector_number,1,EXT));
                (* idt operand to error_code set because vector is used *)
            FI;
            IF software interrupt (* Generated by INT n, INT3, or INTO; does not apply to INT1 *)
                THEN
                    IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
                        THEN #GP(error_code(vector_number,1,0));
                        (* idt operand to error_code set because vector is used *)
                        (* ext operand to error_code is 0 because INT n, INT3, or INTO*)
                    FI;
            FI;
            IF gate not present
                THEN #NP(error_code(vector_number,1,EXT));
                (* idt operand to error_code set because vector is used *)
            FI;
            GOTO TRAP-OR-INTERRUPT-GATE; (* Trap/interrupt gate *)
        ELSE (* CR4.FRED = 1 *)
            FRED event delivery of software interrupt, exception, hardware interrupt, or non-maskable interrupt;
END;

TASK-GATE: (* PE = 1, task gate *)
    Read TSS selector in task gate (IDT descriptor);
        IF local/global bit is set to local or index not within GDT limits
            THEN #GP(error_code(TSS selector,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
        Access TSS descriptor in GDT;
        IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
            THEN #GP(error_code(TSS selector,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
        IF TSS not present
            THEN #NP(error_code(TSS selector,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
    SWITCH-TASKS (with nesting) to TSS;
    IF interrupt caused by fault with error code
        THEN
            IF stack limit does not allow push of error code
                THEN #SS(EXT); FI;
            Push(error code);
    FI;
    IF EIP not within code segment limit
        THEN #GP(EXT); FI;
END;
```

```
TRAP-OR-INTERRUPT-GATE:
    Read new code-segment selector for trap or interrupt gate (IDT descriptor);
    IF new code-segment selector is NULL
        THEN #GP(EXT); FI; (* Error code contains NULL selector *)
    IF new code-segment selector is not within its descriptor table limits
        THEN #GP(error_code(new code-segment selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    Read descriptor referenced by new code-segment selector;
    IF descriptor does not indicate a code segment or new code-segment DPL > CPL
        THEN #GP(error_code(new code-segment selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    IF new code-segment descriptor is not present,
        THEN #NP(error_code(new code-segment selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    IF new code segment is non-conforming with DPL < CPL
        THEN
            IF VM = 0
                THEN
                    GOTO INTER-PRIVILEGE-LEVEL-INTERRUPT;
                    (* PE = 1, VM = 0, interrupt or trap gate, nonconforming code segment,
                    DPL < CPL *)
                ELSE (* VM = 1 *)
                    IF new code-segment DPL ≠ 0
                        THEN #GP(error_code(new code-segment selector,0,EXT));
                        (* idt operand to error_code is 0 because selector is used *)
                    GOTO INTERRUPT-FROM-VIRTUAL-8086-MODE; FI;
                    (* PE = 1, interrupt or trap gate, DPL < CPL, VM = 1 *)
            FI;
        ELSE (* PE = 1, interrupt or trap gate, DPL ≥ CPL *)
            IF VM = 1
                THEN #GP(error_code(new code-segment selector,0,EXT));
                (* idt operand to error_code is 0 because selector is used *)
            IF new code segment is conforming or new code-segment DPL = CPL
                THEN
                    GOTO INTRA-PRIVILEGE-LEVEL-INTERRUPT;
                ELSE (* PE = 1, interrupt or trap gate, nonconforming code segment, DPL > CPL *)
                    #GP(error_code(new code-segment selector,0,EXT));
                    (* idt operand to error_code is 0 because selector is used *)
            FI;
    FI;
END;

INTER-PRIVILEGE-LEVEL-INTERRUPT:
    (* PE = 1, interrupt or trap gate, non-conforming code segment, DPL < CPL *)
    IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
        THEN
        (* Identify stack-segment selector for new privilege level in current TSS *)
            IF current TSS is 32-bit
                THEN
                    TSSstackAddress := (new code-segment DPL « 3) + 4;
                    IF (TSSstackAddress + 5) > current TSS limit
                        THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                        (* idt operand to error_code is 0 because selector is used *)
```

```
                    NewSS := 2 bytes loaded from (TSS base + TSSstackAddress + 4);
                    NewESP := 4 bytes loaded from (TSS base + TSSstackAddress);
                ELSE    (* current TSS is 16-bit *)
                    TSSstackAddress := (new code-segment DPL « 2) + 2
                    IF (TSSstackAddress + 3) > current TSS limit
                        THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                        (* idt operand to error_code is 0 because selector is used *)
                    NewSS := 2 bytes loaded from (TSS base + TSSstackAddress + 2);
                    NewESP := 2 bytes loaded from (TSS base + TSSstackAddress);
            FI;
            IF NewSS is NULL
                THEN #TS(EXT); FI;
            IF NewSS index is not within its descriptor-table limits
            or NewSS RPL ≠ new code-segment DPL
                THEN #TS(error_code(NewSS,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            Read new stack-segment descriptor for NewSS in GDT or LDT;
            IF new stack-segment DPL ≠ new code-segment DPL
            or new stack-segment Type does not indicate writable data segment
                THEN #TS(error_code(NewSS,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            IF NewSS is not present
                THEN #SS(error_code(NewSS,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
                NewSSP := IA32_PLi_SSP (* where i = new code-segment DPL *)
        ELSE (* IA-32e mode *)
            IF IDT-gate IST = 0
                THEN TSSstackAddress := (new code-segment DPL « 3) + 4;
                ELSE TSSstackAddress := (IDT gate IST « 3) + 28;
            FI;
            IF (TSSstackAddress + 7) > current TSS limit
                THEN #TS(error_code(current TSS selector,0,EXT); FI;
                (* idt operand to error_code is 0 because selector is used *)
            NewRSP := 8 bytes loaded from (current TSS base + TSSstackAddress);
            NewSS := new code-segment DPL; (* NULL selector with RPL = new CPL *)
            IF IDT-gate IST = 0
                THEN
                    NewSSP := IA32_PLi_SSP (* where i = new code-segment DPL *)
                ELSE
                    NewSSPAddress = IA32_INTERRUPT_SSP_TABLE_ADDR + (IDT-gate IST « 3)
                    (* Check if shadow stacks are enabled at CPL 0 *)
                    IF ShadowStackEnabled(CPL 0)
                        THEN NewSSP := 8 bytes loaded from NewSSPAddress; FI;
            FI;
    FI;
    IF IDT gate is 32-bit
            THEN
                IF new stack does not have room for 24 bytes (error code pushed)
                or 20 bytes (no error code pushed)
                    THEN #SS(error_code(NewSS,0,EXT)); FI;
                    (* idt operand to error_code is 0 because selector is used *)
            FI
        ELSE
            IF IDT gate is 16-bit
```

```
                THEN
                    IF new stack does not have room for 12 bytes (error code pushed)
                    or 10 bytes (no error code pushed);
                        THEN #SS(error_code(NewSS,0,EXT)); FI;
                        (* idt operand to error_code is 0 because selector is used *)
            ELSE (* 64-bit IDT gate*)
                IF StackAddress is non-canonical
                    THEN #SS(EXT); FI; (* Error code contains NULL selector *)
        FI;
FI;
IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
        THEN
            IF instruction pointer from IDT gate is not within new code-segment limits
                THEN #GP(EXT); FI; (* Error code contains NULL selector *)
            ESP := NewESP;
            SS := NewSS; (* Segment descriptor information also loaded *)
        ELSE (* IA-32e mode *)
            IF instruction pointer from IDT gate contains a non-canonical address
                THEN #GP(EXT); FI; (* Error code contains NULL selector *)
            RSP := NewRSP & FFFFFFFFFFFFFFF0H;
            SS := NewSS;
FI;
IF IDT gate is 32-bit
        THEN
            CS:EIP := Gate(CS:EIP); (* Segment descriptor information also loaded *)
        ELSE
            IF IDT gate 16-bit
                THEN
                    CS:IP := Gate(CS:IP);
                    (* Segment descriptor information also loaded *)
                ELSE (* 64-bit IDT gate *)
                    CS:RIP := Gate(CS:RIP);
                    (* Segment descriptor information also loaded *)
            FI;
FI;
IF IDT gate is 32-bit
            THEN
                Push(far pointer to old stack);
                (* Old SS and ESP, 3 words padded to 4 *)
                Push(EFLAGS);
                Push(far pointer to return instruction);
                (* Old CS and EIP, 3 words padded to 4 *)
                Push(ErrorCode); (* If needed, 4 bytes *)
            ELSE
                IF IDT gate 16-bit
                    THEN
                        Push(far pointer to old stack);
                        (* Old SS and SP, 2 words *)
                        Push(EFLAGS(15:0]);
                        Push(far pointer to return instruction);
                        (* Old CS and IP, 2 words *)
                        Push(ErrorCode); (* If needed, 2 bytes *)
                    ELSE (* 64-bit IDT gate *)
                        Push(far pointer to old stack);
```

```
                        (* Old SS and SP, each an 8-byte push *)
                        Push(RFLAGS); (* 8-byte push *)
                        Push(far pointer to return instruction);
                        (* Old CS and RIP, each an 8-byte push *)
                        Push(ErrorCode); (* If needed, 8-bytes *)
                FI;
    FI;
    IF ShadowStackEnabled(CPL) AND CPL = 3
        THEN
            IF IA32_EFER.LMA = 0
                THEN IA32_PL3_SSP := SSP;
                ELSE (* adjust so bits 63:N get the value of bit N-1, where N is the CPU's maximum linear-address width *)
                    IA32_PL3_SSP := LA_adjust(SSP);
            FI;
    FI;
    CPL := new code-segment DPL;
    CS(RPL) := CPL;
    IF ShadowStackEnabled(CPL)
        oldSSP := SSP
        SSP := NewSSP
        IF SSP & 0x07 != 0
            THEN #GP(0); FI;
        (* Token and CS:LIP:oldSSP pushed on shadow stack must be contained in a naturally aligned 32-byte region *)
        IF (SSP & ~0x1F) != ((SSP - 24) & ~0x1F)
            #GP(0); FI;
        IF ((IA32_EFER.LMA and CS.L) = 0 AND SSP[63:32] != 0)
            THEN #GP(0); FI;
        expected_token_value = SSP              (* busy bit - bit position 0 - must be clear *)
        new_token_value = SSP | BUSY_BIT        (* Set the busy bit *)
        IF shadow_stack_lock_cmpxchg8b(SSP, new_token_value, expected_token_value) != expected_token_value
            THEN #GP(0); FI;
        IF oldSS.DPL != 3
            ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits of 0 *)
            ShadowStackPush8B(oldCSBASE + oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
            ShadowStackPush8B(oldSSP);
        FI;
    FI;
    IF EndbranchEnabled (CPL)
        IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH;
        IA32_S_CET.SUPPRESS = 0
    FI;
    IF IDT gate is interrupt gate
        THEN IF := 0 (* Interrupt flag set to 0, interrupts disabled *); FI;
    TF := 0;
    VM := 0;
    RF := 0;
    NT := 0;
END;

INTERRUPT-FROM-VIRTUAL-8086-MODE:
    (* Identify stack-segment selector for privilege level 0 in current TSS *)
    IF current TSS is 32-bit
        THEN
            IF TSS limit < 9
```

```
                THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            NewSS := 2 bytes loaded from (current TSS base + 8);
            NewESP := 4 bytes loaded from (current TSS base + 4);
        ELSE (* current TSS is 16-bit *)
            IF TSS limit < 5
                THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            NewSS := 2 bytes loaded from (current TSS base + 4);
            NewESP := 2 bytes loaded from (current TSS base + 2);
FI;
IF NewSS is NULL
    THEN #TS(EXT); FI; (* Error code contains NULL selector *)
IF NewSS index is not within its descriptor table limits
or NewSS RPL ≠ 0
    THEN #TS(error_code(NewSS,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
Read new stack-segment descriptor for NewSS in GDT or LDT;
IF new stack-segment DPL ≠ 0 or stack segment does not indicate writable data segment
    THEN #TS(error_code(NewSS,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
IF new stack segment not present
    THEN #SS(error_code(NewSS,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
NewSSP := IA32_PL0_SSP (* the new code-segment DPL must be 0 *)
IF IDT gate is 32-bit
    THEN
        IF new stack does not have room for 40 bytes (error code pushed)
        or 36 bytes (no error code pushed)
            THEN #SS(error_code(NewSS,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
    ELSE (* IDT gate is 16-bit)
        IF new stack does not have room for 20 bytes (error code pushed)
        or 18 bytes (no error code pushed)
            THEN #SS(error_code(NewSS,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
FI;
IF instruction pointer from IDT gate is not within new code-segment limits
    THEN #GP(EXT); FI; (* Error code contains NULL selector *)
tempEFLAGS := EFLAGS;
VM := 0;
TF := 0;
RF := 0;
NT := 0;
IF service through interrupt gate
    THEN IF = 0; FI;
TempSS := SS;
TempESP := ESP;
SS := NewSS;
ESP := NewESP;
(* Following pushes are 16 bits for 16-bit IDT gates and 32 bits for 32-bit IDT gates;
Segment selector pushes in 32-bit mode are padded to two words *)
Push(GS);
Push(FS);
```

```
    Push(DS);
    Push(ES);
    Push(TempSS);
    Push(TempESP);
    Push(TempEFlags);
    Push(CS);
    Push(EIP);
    GS := 0; (* Segment registers made NULL, invalid for use in protected mode *)
    FS := 0;
    DS := 0;
    ES := 0;
    CS := Gate(CS); (* Segment descriptor information also loaded *)
    CS(RPL) := 0;
    CPL := 0;
    IF IDT gate is 32-bit
        THEN
            EIP := Gate(instruction pointer);
        ELSE (* IDT gate is 16-bit *)
            EIP := Gate(instruction pointer) AND 0000FFFFH;
    FI;
    IF ShadowStackEnabled(0)
        oldSSP := SSP
        SSP := NewSSP
        IF SSP & 0x07 != 0
            THEN #GP(0); FI;
        (* Token and CS:LIP:oldSSP pushed on shadow stack must be contained in a naturally aligned 32-byte region *)
        IF (SSP & ~0x1F) != ((SSP – 24) & ~0x1F)
            #GP(0); FI;
    IF ((IA32_EFER.LMA and CS.L) = 0 AND SSP[63:32] != 0)
        THEN #GP(0); FI;
    expected_token_value = SSP (* busy bit - bit position 0 - must be clear *)
    new_token_value = SSP | BUSY_BIT (* Set the busy bit *)
    IF shadow_stack_lock_cmpxchg8b(SSP, new_token_value, expected_token_value) != expected_token_value
        THEN #GP(0); FI;
    FI;
    IF EndbranchEnabled (CPL)
        IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH;
        IA32_S_CET.SUPPRESS = 0
    FI;
(* Start execution of new routine in Protected Mode *)
END;

INTRA-PRIVILEGE-LEVEL-INTERRUPT:
    NewSSP = SSP;
    CHECK_SS_TOKEN = 0
    (* PE = 1, DPL = CPL or conforming segment *)
    IF IA32_EFER.LMA = 1 (* IA-32e mode *)
        IF IDT-descriptor IST ≠ 0
            THEN
                TSSstackAddress := (IDT-descriptor IST « 3) + 28;
                IF (TSSstackAddress + 7) > TSS limit
                    THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                    (* idt operand to error_code is 0 because selector is used *)
                NewRSP := 8 bytes loaded from (current TSS base + TSSstackAddress);
```

```
            ELSE NewRSP := RSP;
        FI;
        IF IDT-descriptor IST ≠ 0
            IF ShadowStackEnabled(CPL)
                THEN
                    NewSSPAddress = IA32_INTERRUPT_SSP_TABLE_ADDR + (IDT gate IST « 3)
                    NewSSP := 8 bytes loaded from NewSSPAddress
                    CHECK_SS_TOKEN = 1
            FI;
        FI;
FI;
IF 32-bit gate (* implies IA32_EFER.LMA = 0 *)
        THEN
            IF current stack does not have room for 16 bytes (error code pushed)
            or 12 bytes (no error code pushed)
                THEN #SS(EXT); FI; (* Error code contains NULL selector *)
        ELSE IF 16-bit gate (* implies IA32_EFER.LMA = 0 *)
            IF current stack does not have room for 8 bytes (error code pushed)
            or 6 bytes (no error code pushed)
                THEN #SS(EXT); FI; (* Error code contains NULL selector *)
        ELSE (* IA32_EFER.LMA = 1, 64-bit gate*)
                IF NewRSP contains a non-canonical address
                    THEN #SS(EXT); (* Error code contains NULL selector *)
        FI;
FI;
IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
        THEN
            IF instruction pointer from IDT gate is not within new code-segment limit
                THEN #GP(EXT); FI; (* Error code contains NULL selector *)
        ELSE
            IF instruction pointer from IDT gate contains a non-canonical address
                THEN #GP(EXT); FI; (* Error code contains NULL selector *)
            RSP := NewRSP & FFFFFFFFFFFFFFF0H;
FI;
IF IDT gate is 32-bit (* implies IA32_EFER.LMA = 0 *)
        THEN
            Push (EFLAGS);
            Push (far pointer to return instruction); (* 3 words padded to 4 *)
            CS:EIP := Gate(CS:EIP); (* Segment descriptor information also loaded *)
            Push (ErrorCode); (* If any *)
        ELSE
            IF IDT gate is 16-bit (* implies IA32_EFER.LMA = 0 *)
                THEN
                    Push (FLAGS);
                    Push (far pointer to return location); (* 2 words *)
                    CS:IP := Gate(CS:IP);
                    (* Segment descriptor information also loaded *)
                    Push (ErrorCode); (* If any *)
                ELSE (* IA32_EFER.LMA = 1, 64-bit gate*)
                    Push(far pointer to old stack);
                    (* Old SS and SP, each an 8-byte push *)
                    Push(RFLAGS); (* 8-byte push *)
                    Push(far pointer to return instruction);
                    (* Old CS and RIP, each an 8-byte push *)
```

```
                    Push(ErrorCode); (* If needed, 8 bytes *)
                    CS:RIP := GATE(CS:RIP);
                    (* Segment descriptor information also loaded *)
            FI;
    FI;
    CS(RPL) := CPL;
    IF ShadowStackEnabled(CPL)
        IF CHECK_SS_TOKEN == 1
            THEN
                IF NewSSP & 0x07 != 0
                    THEN #GP(0); FI;
        (* Token and CS:LIP:oldSSP pushed on shadow stack must be contained in a naturally aligned 32-byte region *)
        IF (NewSSP & ~0x1F) != ((NewSSP – 24) & ~0x1F)
            #GP(0); FI;

                IF ((IA32_EFER.LMA and CS.L) = 0 AND NewSSP[63:32] != 0)
                    THEN #GP(0); FI;
                expected_token_value = NewSSP (* busy bit - bit position 0 - must be clear *)
                new_token_value = NewSSP | BUSY_BIT (* Set the busy bit *)
                IF shadow_stack_lock_cmpxchg8b(NewSSP, new_token_value, expected_token_value) != expected_token_value
                    THEN #GP(0); FI;
        FI;
        (* Align to next 8 byte boundary *)
        tempSSP = SSP;
        Shadow_stack_store 4 bytes of 0 to (NewSSP — 4)
        SSP = newSSP & 0xFFFFFFFFFFFFFFF8H;
        (* push cs:lip:ssp on shadow stack *)
        ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits of 0 *)
        ShadowStackPush8B(oldCSBASE + oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
        ShadowStackPush8B(tempSSP);
    FI;
    IF EndbranchEnabled (CPL)
        IF CPL = 3
            THEN
                IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
                IA32_U_CET.SUPPRESS = 0
            ELSE
                IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                IA32_S_CET.SUPPRESS = 0
        FI;
    FI;
    IF IDT gate is interrupt gate
        THEN IF := 0; FI; (* Interrupt flag set to 0; interrupts disabled *)
    TF := 0;
    NT := 0;
    VM := 0;
    RF := 0;
END;
```

Flags Affected

The EFLAGS register is pushed onto the stack. The IF, TF, NT, AC, RF, and VM flags may be cleared, depending on the mode of operation of the processor when the INT instruction is executed (see the "Operation" section). If the interrupt uses a task gate, any flags may be set or cleared, controlled by the EFLAGS image in the new task's TSS.

Protected Mode Exceptions

#GP(error_code)	If the instruction pointer in the IDT or in the interrupt, trap, or task gate is beyond the code segment limits.
	If the segment selector in the interrupt, trap, or task gate is NULL.
	If an interrupt, trap, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits.
	If the vector selects a descriptor outside the IDT limits.
	If an IDT descriptor is not an interrupt, trap, or task gate.
	If an interrupt is generated by the INT *n*, INT3, or INTO instruction and the DPL of an interrupt, trap, or task gate is less than the CPL.
	If the segment selector in an interrupt or trap gate does not point to a segment descriptor for a code segment.
	If the segment selector for a TSS has its local/global bit set for local.
	If a TSS segment descriptor specifies that the TSS is busy or not available.
	If SSP in IA32_PLi_SSP (where i is the new CPL) is not 8 byte aligned.
	If the token and the stack frame to be pushed on shadow stack are not contained in a naturally aligned 32-byte region of the shadow stack.
	If "supervisor Shadow Stack" token on new shadow stack is marked busy.
	If destination mode is 32-bit or compatibility mode, but SSP address in "supervisor shadow stack" token is beyond 4GB.
	If SSP address in "supervisor shadow stack" token does not match SSP address in IA32_PLi_SSP (where i is the new CPL).
#SS(error_code)	If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment and no stack switch occurs.
	If the SS register is being loaded and the segment pointed to is marked not present.
	If pushing the return address, flags, error code, or stack segment pointer exceeds the bounds of the new stack segment when a stack switch occurs.
#NP(error_code)	If code segment, interrupt gate, trap gate, task gate, or TSS is not present.
#TS(error_code)	If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate.
	If DPL of the stack segment descriptor pointed to by the stack segment selector in the TSS is not equal to the DPL of the code segment descriptor for the interrupt or trap gate.
	If the stack segment selector in the TSS is NULL.
	If the stack segment for the TSS is not a writable data segment.
	If segment-selector index for stack segment is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.
#AC(EXT)	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the interrupt vector number is outside the IDT limits.
#SS	If stack limit violation on push.
	If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(error_code)	(For INT *n,* INTO, or BOUND instruction) If the IOPL is less than 3 or the DPL of the interrupt, trap, or task gate is not equal to 3.

	If the instruction pointer in the IDT or in the interrupt, trap, or task gate is beyond the code segment limits.
	If the segment selector in the interrupt, trap, or task gate is NULL.
	If a interrupt gate, trap gate, task gate, code segment, or TSS segment selector index is outside its descriptor table limits.
	If the vector selects a descriptor outside the IDT limits.
	If an IDT descriptor is not an interrupt, trap, or task gate.
	If an interrupt is generated by INT *n*, INT3, or INTO and the DPL of an interrupt, trap, or task gate is less than the CPL.
	If the segment selector in an interrupt or trap gate does not point to a segment descriptor for a code segment.
	If the segment selector for a TSS has its local/global bit set for local.
#SS(error_code)	If the SS register is being loaded and the segment pointed to is marked not present.
	If pushing the return address, flags, error code, stack segment pointer, or data segments exceeds the bounds of the stack segment.
#NP(error_code)	If code segment, interrupt gate, trap gate, task gate, or TSS is not present.
#TS(error_code)	If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate.
	If DPL of the stack segment descriptor for the TSS's stack segment is not equal to the DPL of the code segment descriptor for the interrupt or trap gate.
	If the stack segment selector in the TSS is NULL.
	If the stack segment for the TSS is not a writable data segment.
	If segment-selector index for stack segment is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.
#OF	If the INTO instruction is executed and the OF flag is set.
#UD	If the LOCK prefix is used.
#AC(EXT)	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

Compatibility Mode Exceptions

#GP(error_code)	If the instruction pointer in the 64-bit interrupt gate or trap gate is non-canonical.
	If the segment selector in the 64-bit interrupt or trap gate is NULL.
	If the vector selects a descriptor outside the IDT limits.
	If the vector points to a gate which is in non-canonical space.
	If the vector points to a descriptor which is not a 64-bit interrupt gate or a 64-bit trap gate.
	If the descriptor pointed to by the gate selector is outside the descriptor table limit.
	If the descriptor pointed to by the gate selector is in non-canonical space.
	If the descriptor pointed to by the gate selector is not a code segment.
	If the descriptor pointed to by the gate selector doesn't have the L-bit set, or has both the L-bit and D-bit set.
	If the descriptor pointed to by the gate selector has DPL > CPL.
#GP(0)	If SSP in IA32_PLi_SSP (where i is the new CPL) is not 8 byte aligned.
	If the token and the stack frame to be pushed on shadow stack are not contained in a naturally aligned 32-byte region of the shadow stack.
	If "supervisor shadow stack" token on new shadow stack is marked busy.
	If destination mode is 32-bit or compatibility mode, but SSP address in "supervisor shadow stack" token is beyond 4GB.
	If SSP address in "supervisor shadow stack" token does not match SSP address in IA32_PLi_SSP (where i is the new CPL).
	If FRED event delivery would load RIP with a non-canonical address.

	If FRED event delivery would load SSP with an address that is not 8-byte aligned.
	If a shadow-stack access performed by FRED event delivery would use an address that is not canonical relative to the current paging mode or cause a LASS violation.
#SS(error_code)	If an ordinary stack access would use an address that is not canonical relative to the current paging mode or cause a LASS violation.
#NP(error_code)	If the 64-bit interrupt-gate, 64-bit trap-gate, or code segment is not present.
#TS(error_code)	If an attempt to load RSP from the TSS causes an access to non-canonical space.
	If the RSP from the TSS is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.
#AC(EXT)	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

64-Bit Mode Exceptions

Same exceptions as in compatibility mode.

INVD—Invalidate Internal Caches

Opcode[1]	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 08	INVD	ZO	Valid	Valid	Flush internal caches; initiate flushing of external caches.

NOTES:

1. See the IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Invalidates (flushes) the processor's internal caches and issues a special-function bus cycle that directs external caches to also flush themselves. Data held in internal caches is not written back to main memory.

After executing this instruction, the processor does not wait for the external caches to complete their flushing operation before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache flush signal.

The INVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction.

The INVD instruction may be used when the cache is used as temporary memory and the cache contents need to be invalidated rather than written back to memory. When the cache is used as temporary memory, no external device should be actively writing data to main memory.

Use this instruction with care. Data cached internally and not written back to main memory will be lost. Note that any data from an external device to main memory (for example, via a PCIWrite) can be temporarily stored in the caches; these data can be lost when an INVD instruction is executed. Unless there is a specific requirement or benefit to flushing caches without writing back modified cache lines (for example, temporary memory, testing, or fault recovery where cache coherency with main memory is not a concern), software should instead use the WBINVD instruction.

On processors that support processor reserved memory, the INVD instruction cannot be executed when processor reserved memory protections are activated. See Section 39.5, "EPC and Management of EPC Pages," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D.

On processors that support SEAM, the INVD instruction cannot be executed when the SEAM range is protected. See Section 35.4, "Memory Protection," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3.

Some processors prevent execution of INVD after BIOS execution is complete. They report this by enumerating CPUID.07H.01H:EAX[30] as 1. On such processors, INVD cannot be executed if bit 0 of SR_BIOS_DONE (MSR address 151H) is 1.

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

IA-32 Architecture Compatibility

The INVD instruction is implementation dependent; it may be implemented differently on different families of Intel 64 or IA-32 processors. This instruction is not supported on IA-32 processors earlier than the Intel486 processor.

Operation

```
Flush(InternalCaches);
SignalFlush(ExternalCaches);
Continue (* Continue execution *)
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If the processor reserved memory protections are activated.
	If the SEAM range is protected.
	If CPUID.07H.01H:EAX[30] = 1 and bit 0 is set in MSR_BIOS_DONE (MSR address 151H).
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP(0)	If CPUID.07H.01H:EAX[30] = 1 and bit 0 is set in MSR_BIOS_DONE (MSR address 151H).
	If the processor reserved memory protections are activated.
	If the SEAM range is protected.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	The INVD instruction cannot be executed in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

INVLPG—Invalidate TLB Entries

Opcode[1]	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01/7	INVLPG m	M	Valid	Valid	Invalidate TLB entries for page containing m.

NOTES:

1. See the IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

Description

Invalidates any translation lookaside buffer (TLB) entries specified with the source operand. The source operand is a memory address. The processor determines the page that contains that address and flushes all TLB entries for that page.[1]

The INVLPG instruction is a privileged instruction. When the processor is running in protected mode, the CPL must be 0 to execute this instruction.

The INVLPG instruction normally flushes TLB entries only for the specified page; however, in some cases, it may flush more entries, even the entire TLB. The instruction invalidates TLB entries associated with the current PCID and may or may not do so for TLB entries associated with other PCIDs. (If PCIDs are disabled — CR4.PCIDE = 0 — the current PCID is 000H.) The instruction also invalidates any global TLB entries for the specified page, regardless of PCID.

For more details on operations that flush the TLB, see "MOV—Move to/from Control Registers" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B, and Section 5.10.4.1, "Operations that Invalidate TLBs and Paging-Structure Caches," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

This instruction's operation is the same in all non-64-bit modes. It also operates the same in 64-bit mode, except if the memory address is in non-canonical form. In this case, INVLPG is the same as a NOP.

IA-32 Architecture Compatibility

The INVLPG instruction is implementation dependent, and its function may be implemented differently on different families of Intel 64 or IA-32 processors. This instruction is not supported on IA-32 processors earlier than the Intel486 processor.

Operation

```
Invalidate(RelevantTLBEntries);
Continue; (* Continue execution *)
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	Operand is a register.
	If the LOCK prefix is used.

1. If the paging structures map the linear address using a page larger than 4 KBytes and there are multiple TLB entries for that page (see Section 5.10.2.3, "Details of TLB Use," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A), the instruction invalidates all of them.

Real-Address Mode Exceptions

#UD	Operand is a register. If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	The INVLPG instruction cannot be executed at the virtual-8086 mode.

64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	Operand is a register. If the LOCK prefix is used.

INVPCID—Invalidate Process-Context Identifier

Opcode/Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 82 /r INVPCID r32, m128	RM	N.E./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	RM	V/N.E.	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
RM	ModRM:reg (r)	ModRM:r/m (r)	N/A	N/A

Description

Invalidates mappings in the translation lookaside buffers (TLBs) and paging-structure caches based on process-context identifier (PCID). (See Section 5.10, "Caching Translation Information," in the *Intel 64 and IA-32 Architecture Software Developer's Manual, Volume 3A*.) Invalidation is based on the INVPCID type specified in the register operand and the INVPCID descriptor specified in the memory operand.

Outside 64-bit 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.

There are four INVPCID types currently defined:

- Individual-address invalidation: If the INVPCID type is 0, the logical processor invalidates mappings—except global translations—for the linear address and PCID specified in the INVPCID descriptor.[1] In some cases, the instruction may invalidate global translations or mappings for other linear addresses (or other PCIDs) as well.
- Single-context invalidation: If the INVPCID type is 1, the logical processor invalidates all mappings—except global translations—associated with the PCID specified in the INVPCID descriptor. In some cases, the instruction may invalidate global translations or mappings for other PCIDs as well.
- All-context invalidation, including global translations: If the INVPCID type is 2, the logical processor invalidates all mappings—including global translations—associated with any PCID.
- All-context invalidation: If the INVPCID type is 3, the logical processor invalidates all mappings—except global translations—associated with any PCID. In some case, the instruction may invalidate global translations as well.

The INVPCID descriptor comprises 128 bits and consists of a PCID and a linear address as shown in Figure 3-20. For INVPCID type 0, the processor uses the full 64 bits of the linear address even outside 64-bit mode; the linear address is not used for other INVPCID types.

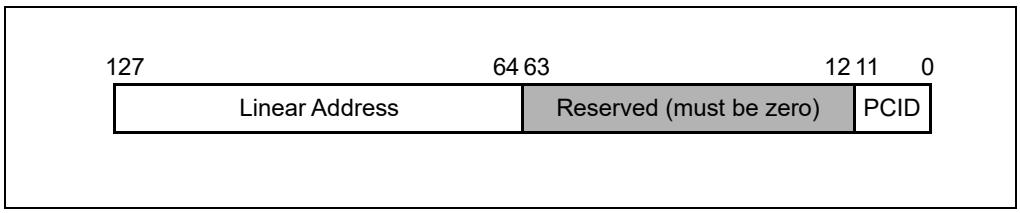


Figure 3-20. INVPCID Descriptor

1. If the paging structures map the linear address using a page larger than 4 KBytes and there are multiple TLB entries for that page (see Section 5.10.2.3, "Details of TLB Use," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A), the instruction invalidates all of them.

If CR4.PCIDE = 0, a logical processor does not cache information for any PCID other than 000H. In this case, executions with INVPCID types 0 and 1 are allowed only if the PCID specified in the INVPCID descriptor is 000H; executions with INVPCID types 2 and 3 invalidate mappings only for PCID 000H. Note that CR4.PCIDE must be 0 outside IA-32e mode (see Section 5.10.1, "Process-Context Identifiers (PCIDs)," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A).

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
        PCID := INVPCID_DESC[11:0];
        L_ADDR := INVPCID_DESC[127:64];
        Invalidate mappings for L_ADDR associated with PCID except global translations;
        BREAK;
    1:                  // single PCID invalidation
        PCID := INVPCID_DESC[11:0];
        Invalidate all mappings associated with PCID except global translations;
        BREAK;
    2:                  // all PCID invalidation including global translations
        Invalidate all mappings for all PCIDs, including global translations;
        BREAK;
    3:                  // all PCID invalidation retaining global translations
        Invalidate all mappings for all PCIDs 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 INVPCID_TYPE is either 0 or 1 and INVPCID_DESC[11:0] is not zero.
	If INVPCID_TYPE is 0 and the linear address in INVPCID_DESC[127:64] is not canonical.
#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.07H.00H:EBX.INVPCID[10] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	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 INVPCID_TYPE is either 0 or 1 and INVPCID_DESC[11:0] is not zero.
	If INVPCID_TYPE is 0 and the linear address in INVPCID_DESC[127:64] is not canonical.
#UD	If CPUID.07H.00H:EBX.INVPCID[10] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	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, i.e., INVPCID_TYPE > 3.
	If bits 63:12 of INVPCID_DESC are not all zero.
	If CR4.PCIDE=0, INVPCID_TYPE is either 0 or 1, and INVPCID_DESC[11:0] is not zero.
	If INVPCID_TYPE is 0 and the linear address in INVPCID_DESC[127:64] is not canonical.
#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.07H.00H:EBX.INVPCID[10] = 0.

IRET/IRETD/IRETQ—Interrupt Return

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
CF	IRET	ZO	Valid	Valid	Interrupt return (16-bit operand size).
CF	IRETD	ZO	Valid	Valid	Interrupt return (32-bit operand size).
REX.W + CF	IRETQ	ZO	Valid	N.E.	Interrupt return (64-bit operand size).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Returns program control from an exception or interrupt handler to a program or procedure that was interrupted by an exception, an external interrupt, or a software-generated interrupt. These instructions are also used to perform a return from a nested task. (A nested task is created when a CALL instruction is used to initiate a task switch or when an interrupt or exception causes a task switch to an interrupt or exception handler.) See the section titled "Task Linking" in Chapter 10 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

IRET and IRETD are mnemonics for the same opcode. The IRETD mnemonic (interrupt return double) is intended for use when returning from an interrupt when using the 32-bit operand size; however, most assemblers use the IRET mnemonic interchangeably for both operand sizes.

In Real-Address Mode, the IRET instruction performs a far return to the interrupted program or procedure. During this operation, the processor pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure.

In Protected Mode, the action of the IRET instruction depends on the settings of the NT (nested task) and VM flags in the EFLAGS register and the VM flag in the EFLAGS image stored on the current stack. Depending on the setting of these flags, the processor performs the following types of interrupt returns:

- Return from virtual-8086 mode.
- Return to virtual-8086 mode.
- Intra-privilege level return.
- Inter-privilege level return.
- Return from nested task (task switch).

If the NT flag (EFLAGS register) is cleared, the IRET instruction performs a far return from the interrupt procedure, without a task switch. The code segment being returned to must be equally or less privileged than the interrupt handler routine (as indicated by the RPL field of the code segment selector popped from the stack).

As with a real-address mode interrupt return, the IRET instruction pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure. If the return is to another privilege level, the IRET instruction also pops the stack pointer and SS from the stack, before resuming program execution. If the return is to virtual-8086 mode, the processor also pops the data segment registers from the stack.

If the NT flag is set, the IRET instruction performs a task switch (return) from a nested task (a task called with a CALL instruction, an interrupt, or an exception) back to the calling or interrupted task. The updated state of the task executing the IRET instruction is saved in its TSS. If the task is re-entered later, the code that follows the IRET instruction is executed.

If the NT flag is set and the processor is in IA-32e mode, the IRET instruction causes a general protection exception.

If nonmaskable interrupts (NMIs) are blocked (see Section 7.7.2, "NMI Blocking" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A), execution of the IRET instruction unblocks NMIs. This

unblocking occurs even if the instruction causes a fault. In such a case, NMIs are unmasked before the exception handler is invoked.

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

Refer to Chapter 6, "Procedure Calls, Interrupts, and Exceptions" and Chapter 18, "Control-flow Enforcement Technology (CET)" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for CET details.

When FRED transitions are enabled, an execution of IRET that would change CPL causes a general-protection exception, as does an execution of IRET that would enter compatibility mode when CPL is 0.

Instruction ordering. IRET is a serializing instruction. See Section 11.3 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 28 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C, for more information about the behavior of this instruction in VMX non-root operation.

Operation

```
IF PE = 0
    THEN GOTO REAL-ADDRESS-MODE;
ELSIF (IA32_EFER.LMA = 0)
    THEN
        IF (EFLAGS.VM = 1)
            THEN GOTO RETURN-FROM-VIRTUAL-8086-MODE;
            ELSE GOTO PROTECTED-MODE;
        FI;
    ELSE GOTO IA-32e-MODE;
FI;

REAL-ADDRESS-MODE;
    IF OperandSize = 32
        THEN
            EIP := Pop();
            CS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            tempEFLAGS := Pop();
            EFLAGS := (tempEFLAGS AND 257FD5H) OR (EFLAGS AND 1A0000H);
        ELSE (* OperandSize = 16 *)
            EIP := Pop(); (* 16-bit pop; clear upper 16 bits *)
            CS := Pop(); (* 16-bit pop *)
            EFLAGS[15:0] := Pop();
    FI;
    END;

RETURN-FROM-VIRTUAL-8086-MODE:
(* Processor is in virtual-8086 mode when IRET is executed and stays in virtual-8086 mode *)
    IF IOPL = 3 (* Virtual mode: PE = 1, VM = 1, IOPL = 3 *)
        THEN IF OperandSize = 32
            THEN
                EIP := Pop();
                CS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
                EFLAGS := Pop();
                (* VM, IOPL,VIP and VIF EFLAG bits not modified by pop *)
                IF EIP not within CS limit
                    THEN #GP(0); FI;
            ELSE (* OperandSize = 16 *)
                EIP := Pop(); (* 16-bit pop; clear upper 16 bits *)
```

```
                CS := Pop(); (* 16-bit pop *)
                EFLAGS[15:0] := Pop(); (* IOPL in EFLAGS not modified by pop *)
                IF EIP not within CS limit
                    THEN #GP(0); FI;
            FI;
        ELSE
            #GP(0); (* Trap to virtual-8086 monitor: PE = 1, VM = 1, IOPL < 3 *)
    FI;
END;

PROTECTED-MODE:
    IF NT = 1
        THEN GOTO TASK-RETURN; (* PE = 1, VM = 0, NT = 1 *)
    FI;
    IF OperandSize = 32
        THEN
            EIP := Pop();
            CS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            tempEFLAGS := Pop();
        ELSE (* OperandSize = 16 *)
            EIP := Pop(); (* 16-bit pop; clear upper bits *)
            CS := Pop(); (* 16-bit pop *)
            tempEFLAGS := Pop(); (* 16-bit pop; clear upper bits *)
    FI;
    IF tempEFLAGS(VM) = 1 and CPL = 0
        THEN GOTO RETURN-TO-VIRTUAL-8086-MODE;
        ELSE GOTO PROTECTED-MODE-RETURN;
    FI;

TASK-RETURN: (* PE = 1, VM = 0, NT = 1 *)
    SWITCH-TASKS (without nesting) to TSS specified in link field of current TSS;
    Mark the task just abandoned as NOT BUSY;
    IF EIP is not within CS limit
        THEN #GP(0); FI;
END;

RETURN-TO-VIRTUAL-8086-MODE:
    (* Interrupted procedure was in virtual-8086 mode: PE = 1, CPL=0, VM = 1 in flag image *)
    (* If shadow stack or indirect branch tracking at CPL3 then #GP(0) *)
    IF CR4.CET AND (IA32_U_CET.ENDBR_EN OR IA32_U_CET.SHSTK_EN)
        THEN #GP(0); FI;
    shadowStackEnabled = ShadowStackEnabled(CPL)
    IF EIP not within CS limit
        THEN #GP(0); FI;
    EFLAGS := tempEFLAGS;
    ESP := Pop();
    SS := Pop(); (* Pop 2 words; throw away high-order word *)
    ES := Pop(); (* Pop 2 words; throw away high-order word *)
    DS := Pop(); (* Pop 2 words; throw away high-order word *)
    FS := Pop(); (* Pop 2 words; throw away high-order word *)
    GS := Pop(); (* Pop 2 words; throw away high-order word *)
    IF shadowStackEnabled
        (* check if 8 byte aligned *)
        IF SSP AND 0x7 != 0
```

```
                THEN #CP(FAR-RET/IRET); FI;
    FI;

    CPL := 3;
    (* Resume execution in Virtual-8086 mode *)
    tempOldSSP = SSP;
    (* Now past all faulting points; safe to free the token. The token free is done using the old SSP
     * and using a supervisor override as old CPL was a supervisor privilege level *)
    IF shadowStackEnabled
        expected_token_value = tempOldSSP | BUSY_BIT     (* busy bit - bit position 0 - must be set *)
        new_token_value = tempOldSSP                     (* clear the busy bit *)
        shadow_stack_lock_cmpxchg8b(tempOldSSP, new_token_value, expected_token_value)
    FI;
END;

PROTECTED-MODE-RETURN: (* PE = 1 *)
    IF CS(RPL) > CPL
        THEN GOTO RETURN-TO-OUTER-PRIVILEGE-LEVEL;
        ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; FI;
END;

RETURN-TO-OUTER-PRIVILEGE-LEVEL:
    IF OperandSize = 32
        THEN
            tempESP := Pop();
            tempSS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
    ELSE IF OperandSize = 16
        THEN
            tempESP := Pop(); (* 16-bit pop; clear upper bits *)
            tempSS := Pop(); (* 16-bit pop *)
        ELSE (* OperandSize = 64 *)
            tempRSP := Pop();
            tempSS := Pop(); (* 64-bit pop, high-order 48 bits discarded *)
    FI;
    IF new mode ≠ 64-Bit Mode
        THEN
            IF EIP is not within CS limit
                THEN #GP(0); FI;
        ELSE (* new mode = 64-bit mode *)
            IF RIP is non-canonical
                    THEN #GP(0); FI;
    FI;
    EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) := tempEFLAGS;
    IF OperandSize = 32 or OperandSize = 64
        THEN EFLAGS(RF, AC, ID) := tempEFLAGS; FI;
    IF CPL ≤ IOPL
        THEN EFLAGS(IF) := tempEFLAGS; FI;
    IF CPL = 0
        THEN
            EFLAGS(IOPL) := tempEFLAGS;
            IF OperandSize = 32 or OperandSize = 64
                THEN EFLAGS(VIF, VIP) := tempEFLAGS; FI;
    FI;
    IF ShadowStackEnabled(CPL)
```

```
        (* check if 8 byte aligned *)
        IF SSP AND 0x7 != 0
            THEN #CP(FAR-RET/IRET); FI;
        IF CS(RPL) != 3
            THEN
                tempSsCS = shadow_stack_load 8 bytes from SSP+16;
                tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
                tempSSP = shadow_stack_load 8 bytes from SSP;
                SSP = SSP + 24;
                (* Do 64 bit compare to detect bits beyond 15 being set *)
                tempCS = CS; (* zero padded to 64 bit *)
                IF tempCS != tempSsCS
                    THEN #CP(FAR-RET/IRET); FI;
                (* Do 64 bit compare; pad CSBASE+RIP with 0 for 32 bit LIP *)
                IF CSBASE + RIP != tempSsEIP
                    THEN #CP(FAR-RET/IRET); FI;
                (* check if 4 byte aligned *)
                IF tempSSP AND 0x3 != 0
                    THEN #CP(FAR-RET/IRET); FI;
        FI;
    FI;
    tempOldCPL = CPL;
    CPL := CS(RPL);
        IF OperandSize = 64
            THEN
                RSP := tempRSP;
                SS := tempSS;
        ELSE
            ESP := tempESP;
            SS := tempSS;
        FI;
        IF new mode != 64-Bit Mode
            THEN
                IF EIP is not within CS limit
                    THEN #GP(0); FI;
        ELSE (* new mode = 64-bit mode *)
            IF RIP is non-canonical
                THEN #GP(0); FI;
        FI;
        tempOldSSP = SSP;
        IF ShadowStackEnabled(CPL)
            IF CPL = 3
                THEN tempSSP := IA32_PL3_SSP; FI;
        IF ((IA32_EFER.LMA AND CS.L) = 0 AND tempSSP[63:32] != 0) OR
           ((IA32_EFER.LMA AND CS.L) = 1 AND tempSSP is not canonical relative to the current paging mode)
            THEN #GP(0); FI;
        SSP := tempSSP
        FI;
        (* Now past all faulting points; safe to free the token. The token free is done using the old SSP
         * and using a supervisor override as old CPL was a supervisor privilege level *)
        IF ShadowStackEnabled(tempOldCPL)
            expected_token_value = tempOldSSP | BUSY_BIT (* busy bit - bit position 0 - must be set *)
            new_token_value = tempOldSSP (* clear the busy bit *)
            shadow_stack_lock_cmpxchg8b(tempOldSSP, new_token_value, expected_token_value)
```

```
        FI;

    FOR each SegReg in (ES, FS, GS, and DS)
        DO
            tempDesc := descriptor cache for SegReg (* hidden part of segment register *)
            IF (SegmentSelector == NULL) OR (tempDesc(DPL) < CPL AND tempDesc(Type) is (data or non-conforming code)))
                THEN (* Segment register invalid *)
                    SegmentSelector := 0; (*Segment selector becomes null*)
            FI;
        OD;
END;

RETURN-TO-SAME-PRIVILEGE-LEVEL: (* PE = 1, RPL = CPL *)
    IF new mode ≠ 64-Bit Mode
        THEN
            IF EIP is not within CS limit
                THEN #GP(0); FI;
        ELSE (* new mode = 64-bit mode *)
            IF RIP is non-canonical
                    THEN #GP(0); FI;
    FI;
    EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) := tempEFLAGS;
    IF OperandSize = 32 or OperandSize = 64
        THEN EFLAGS(RF, AC, ID) := tempEFLAGS; FI;
    IF CPL ≤ IOPL
        THEN EFLAGS(IF) := tempEFLAGS; FI;
    IF CPL = 0
         THEN
             EFLAGS(IOPL) := tempEFLAGS;
             IF OperandSize = 32 or OperandSize = 64
                 THEN EFLAGS(VIF, VIP) := tempEFLAGS; FI;
    FI;
    IF ShadowStackEnabled(CPL)
        IF SSP AND 0x7 != 0 (* check if aligned to 8 bytes *)
            THEN #CP(FAR-RET/IRET); FI;
        tempSsCS = shadow_stack_load 8 bytes from SSP+16;
        tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
        tempSSP = shadow_stack_load 8 bytes from SSP;
        SSP = SSP + 24;
        tempCS = CS; (* zero padded to 64 bit *)
        IF tempCS != tempSsCS (* 64 bit compare; CS zero padded to 64 bits *)
            THEN #CP(FAR-RET/IRET); FI;
        IF CSBASE + RIP != tempSsLIP (* 64 bit compare; CSBASE+RIP zero padded to 64 bit for 32 bit LIP *)
            THEN #CP(FAR-RET/IRET); FI;
        IF tempSSP AND 0x3 != 0 (* check if aligned to 4 bytes *)
            THEN #CP(FAR-RET/IRET); FI;
        IF ((IA32_EFER.LMA AND CS.L) = 0 AND tempSSP[63:32] != 0) OR
           ((IA32_EFER.LMA AND CS.L) = 1 AND tempSSP is not canonical relative to the current paging mode)
            THEN #GP(0); FI;
    FI;
    IF ShadowStackEnabled(CPL)
        IF IA32_EFER.LMA = 1
        (* In IA-32e-mode the IRET may be switching stacks if the interrupt/exception was delivered
         through an IDT with a non-zero IST *)
```

```
        (* In IA-32e mode for same CPL IRET there is always a stack switch. The below check verifies if the
         stack switch was to self stack and if so, do not try to free the token on this shadow stack. If the
         tempSSP was not to same stack then there was a stack switch so do attempt to free the token *)
            IF tempSSP != SSP
                THEN
                    expected_token_value = SSP | BUSY_BIT          (* busy bit - bit position 0 - must be set *)
                    new_token_value = SSP                          (* clear the busy bit *)
                    shadow_stack_lock_cmpxchg8b(SSP, new_token_value, expected_token_value)
            FI;
        FI;
        SSP := tempSSP
    FI;
END;

IA-32e-MODE:
    IF NT = 1
        THEN #GP(0);
    ELSE IF OperandSize = 32
        THEN
            EIP := Pop();
            CS := Pop();
            tempEFLAGS := Pop();
        ELSE IF OperandSize = 16
            THEN
                EIP := Pop(); (* 16-bit pop; clear upper bits *)
                CS := Pop(); (* 16-bit pop *)
                tempEFLAGS := Pop(); (* 16-bit pop; clear upper bits *)
            FI;
        ELSE (* OperandSize = 64 *)
            THEN
                    RIP := Pop();
                    CS := Pop(); (* 64-bit pop, high-order 48 bits discarded *)
                    tempRFLAGS := Pop();
    FI;
    IF CS.RPL < CPL or (CR4.FRED = 1 and CS.RPL > CPL)
        THEN #GP(CS.selector); FI;
    IF CS.RPL > CPL
        THEN GOTO RETURN-TO-OUTER-PRIVILEGE-LEVEL;
        ELSE    (* CS.RPL = CPL *)
            IF CR4.FRED = 1 and CPL = 0 and CS.L = 0
                THEN #GP(CS.selector); FI;
            IF instruction began in 64-Bit Mode
                THEN
                    IF OperandSize = 32
                        THEN
                            ESP := Pop();
                            SS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
                    ELSE IF OperandSize = 16
                        THEN
                            ESP := Pop(); (* 16-bit pop; clear upper bits *)
                            SS := Pop(); (* 16-bit pop *)
                    ELSE (* OperandSize = 64 *)
                            RSP := Pop();
                            SS := Pop(); (* 64-bit pop, high-order 48 bits discarded *)
```

```
            FI;
        FI;
        GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; FI;
END;
```

Flags Affected

All the flags and fields in the EFLAGS register are potentially modified, depending on the mode of operation of the processor. If performing a return from a nested task to a previous task, the EFLAGS register will be modified according to the EFLAGS image stored in the previous task's TSS.

Protected Mode Exceptions

#GP(0)	If the return code or stack segment selector is NULL.
	If the return instruction pointer is not within the return code segment limit.
#GP(selector)	If a segment selector index is outside its descriptor table limits.
	If the return code segment selector RPL is less than the CPL.
	If the DPL of a conforming-code segment is greater than the return code segment selector RPL.
	If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.
	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
	If the stack segment is not a writable data segment.
	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
	If the segment descriptor for a code segment does not indicate it is a code segment.
	If the segment selector for a TSS has its local/global bit set for local.
	If a TSS segment descriptor specifies that the TSS is not busy.
	If a TSS segment descriptor specifies that the TSS is not available.
#SS(0)	If the top bytes of stack are not within stack limits.
	If the return stack segment is not present.
#NP (selector)	If the return code segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.
#CP (Far-RET/IRET)	If the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is not 4 byte aligned.
	If returning to 32-bit or compatibility mode and the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is beyond 4GB.
	If return instruction pointer from stack and shadow stack do not match.

Real-Address Mode Exceptions

#GP	If the return instruction pointer is not within the return code segment limit.
#SS	If the top bytes of stack are not within stack limits.

Virtual-8086 Mode Exceptions

#GP(0)	If the return instruction pointer is not within the return code segment limit.
	IF IOPL not equal to 3.
#PF(fault-code)	If a page fault occurs.
#SS(0)	If the top bytes of stack are not within stack limits.
#AC(0)	If an unaligned memory reference occurs and alignment checking is enabled.

#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

#GP(0)	If EFLAGS.NT[bit 14] = 1.

Other exceptions same as in Protected Mode.

64-Bit Mode Exceptions

#GP(0)	If EFLAGS.NT[bit 14] = 1.
	If the return code segment selector is NULL.
	If the stack segment selector is NULL going back to compatibility mode.
	If the stack segment selector is NULL going back to CPL3 64-bit mode.
	If a NULL stack segment selector RPL is not equal to CPL going back to non-CPL3 64-bit mode.
	If the return instruction pointer is not within the return code segment limit.
	If the return instruction pointer is non-canonical.
#GP(Selector)	If a segment selector index is outside its descriptor table limits.
	If a segment descriptor memory address is non-canonical.
	If the segment descriptor for a code segment does not indicate it is a code segment.
	If the proposed new code segment descriptor has both the D-bit and L-bit set.
	If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.
	If CPL is greater than the RPL of the code segment selector.
	If FRED transitions are enabled and CPL is less than the RPL of the code segment selector.
	If FRED transitions are enabled and CPL = 0 and IRET would enter compatibility mode.
	If the DPL of a conforming-code segment is greater than the return code segment selector RPL.
	If the stack segment is not a writable data segment.
	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
#SS(0)	If an attempt to pop a value off the stack violates the SS limit.
	If an attempt to pop a value off the stack causes a non-canonical address to be referenced.
	If the return stack segment is not present.
#NP (selector)	If the return code segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.
#CP (Far-RET/IRET)	If the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is not 4 byte aligned.
	If returning to 32-bit or compatibility mode and the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is beyond 4GB.
	If return instruction pointer from stack and shadow stack do not match.

Jcc—Jump if Condition Is Met

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
77 cb	JA rel8	D	Valid	Valid	Jump short if above (CF=0 and ZF=0).
73 cb	JAE rel8	D	Valid	Valid	Jump short if above or equal (CF=0).
72 cb	JB rel8	D	Valid	Valid	Jump short if below (CF=1).
76 cb	JBE rel8	D	Valid	Valid	Jump short if below or equal (CF=1 or ZF=1).
72 cb	JC rel8	D	Valid	Valid	Jump short if carry (CF=1).
E3 cb	JCXZ rel8	D	N.E.	Valid	Jump short if CX register is 0.
E3 cb	JECXZ rel8	D	Valid	Valid	Jump short if ECX register is 0.
E3 cb	JRCXZ rel8	D	Valid	N.E.	Jump short if RCX register is 0.
74 cb	JE rel8	D	Valid	Valid	Jump short if equal (ZF=1).
7F cb	JG rel8	D	Valid	Valid	Jump short if greater (ZF=0 and SF=OF).
7D cb	JGE rel8	D	Valid	Valid	Jump short if greater or equal (SF=OF).
7C cb	JL rel8	D	Valid	Valid	Jump short if less (SF≠ OF).
7E cb	JLE rel8	D	Valid	Valid	Jump short if less or equal (ZF=1 or SF≠ OF).
76 cb	JNA rel8	D	Valid	Valid	Jump short if not above (CF=1 or ZF=1).
72 cb	JNAE rel8	D	Valid	Valid	Jump short if not above or equal (CF=1).
73 cb	JNB rel8	D	Valid	Valid	Jump short if not below (CF=0).
77 cb	JNBE rel8	D	Valid	Valid	Jump short if not below or equal (CF=0 and ZF=0).
73 cb	JNC rel8	D	Valid	Valid	Jump short if not carry (CF=0).
75 cb	JNE rel8	D	Valid	Valid	Jump short if not equal (ZF=0).
7E cb	JNG rel8	D	Valid	Valid	Jump short if not greater (ZF=1 or SF≠ OF).
7C cb	JNGE rel8	D	Valid	Valid	Jump short if not greater or equal (SF≠ OF).
7D cb	JNL rel8	D	Valid	Valid	Jump short if not less (SF=OF).
7F cb	JNLE rel8	D	Valid	Valid	Jump short if not less or equal (ZF=0 and SF=OF).
71 cb	JNO rel8	D	Valid	Valid	Jump short if not overflow (OF=0).
7B cb	JNP rel8	D	Valid	Valid	Jump short if not parity (PF=0).
79 cb	JNS rel8	D	Valid	Valid	Jump short if not sign (SF=0).
75 cb	JNZ rel8	D	Valid	Valid	Jump short if not zero (ZF=0).
70 cb	JO rel8	D	Valid	Valid	Jump short if overflow (OF=1).
7A cb	JP rel8	D	Valid	Valid	Jump short if parity (PF=1).
7A cb	JPE rel8	D	Valid	Valid	Jump short if parity even (PF=1).
7B cb	JPO rel8	D	Valid	Valid	Jump short if parity odd (PF=0).
78 cb	JS rel8	D	Valid	Valid	Jump short if sign (SF=1).
74 cb	JZ rel8	D	Valid	Valid	Jump short if zero (ZF = 1).
0F 87 cw	JA rel16	D	N.S.	Valid	Jump near if above (CF=0 and ZF=0). Not supported in 64-bit mode.
0F 87 cd	JA rel32	D	Valid	Valid	Jump near if above (CF=0 and ZF=0).
0F 83 cw	JAE rel16	D	N.S.	Valid	Jump near if above or equal (CF=0). Not supported in 64-bit mode.
0F 83 cd	JAE rel32	D	Valid	Valid	Jump near if above or equal (CF=0).

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 82 cw	JB rel16	D	N.S.	Valid	Jump near if below (CF=1). Not supported in 64-bit mode.
0F 82 cd	JB rel32	D	Valid	Valid	Jump near if below (CF=1).
0F 86 cw	JBE rel16	D	N.S.	Valid	Jump near if below or equal (CF=1 or ZF=1). Not supported in 64-bit mode.
0F 86 cd	JBE rel32	D	Valid	Valid	Jump near if below or equal (CF=1 or ZF=1).
0F 82 cw	JC rel16	D	N.S.	Valid	Jump near if carry (CF=1). Not supported in 64-bit mode.
0F 82 cd	JC rel32	D	Valid	Valid	Jump near if carry (CF=1).
0F 84 cw	JE rel16	D	N.S.	Valid	Jump near if equal (ZF=1). Not supported in 64-bit mode.
0F 84 cd	JE rel32	D	Valid	Valid	Jump near if equal (ZF=1).
0F 84 cw	JZ rel16	D	N.S.	Valid	Jump near if 0 (ZF=1). Not supported in 64-bit mode.
0F 84 cd	JZ rel32	D	Valid	Valid	Jump near if 0 (ZF=1).
0F 8F cw	JG rel16	D	N.S.	Valid	Jump near if greater (ZF=0 and SF=OF). Not supported in 64-bit mode.
0F 8F cd	JG rel32	D	Valid	Valid	Jump near if greater (ZF=0 and SF=OF).
0F 8D cw	JGE rel16	D	N.S.	Valid	Jump near if greater or equal (SF=OF). Not supported in 64-bit mode.
0F 8D cd	JGE rel32	D	Valid	Valid	Jump near if greater or equal (SF=OF).
0F 8C cw	JL rel16	D	N.S.	Valid	Jump near if less (SF≠ OF). Not supported in 64-bit mode.
0F 8C cd	JL rel32	D	Valid	Valid	Jump near if less (SF≠ OF).
0F 8E cw	JLE rel16	D	N.S.	Valid	Jump near if less or equal (ZF=1 or SF≠ OF). Not supported in 64-bit mode.
0F 8E cd	JLE rel32	D	Valid	Valid	Jump near if less or equal (ZF=1 or SF≠ OF).
0F 86 cw	JNA rel16	D	N.S.	Valid	Jump near if not above (CF=1 or ZF=1). Not supported in 64-bit mode.
0F 86 cd	JNA rel32	D	Valid	Valid	Jump near if not above (CF=1 or ZF=1).
0F 82 cw	JNAE rel16	D	N.S.	Valid	Jump near if not above or equal (CF=1). Not supported in 64-bit mode.
0F 82 cd	JNAE rel32	D	Valid	Valid	Jump near if not above or equal (CF=1).
0F 83 cw	JNB rel16	D	N.S.	Valid	Jump near if not below (CF=0). Not supported in 64-bit mode.
0F 83 cd	JNB rel32	D	Valid	Valid	Jump near if not below (CF=0).
0F 87 cw	JNBE rel16	D	N.S.	Valid	Jump near if not below or equal (CF=0 and ZF=0). Not supported in 64-bit mode.
0F 87 cd	JNBE rel32	D	Valid	Valid	Jump near if not below or equal (CF=0 and ZF=0).
0F 83 cw	JNC rel16	D	N.S.	Valid	Jump near if not carry (CF=0). Not supported in 64-bit mode.
0F 83 cd	JNC rel32	D	Valid	Valid	Jump near if not carry (CF=0).
0F 85 cw	JNE rel16	D	N.S.	Valid	Jump near if not equal (ZF=0). Not supported in 64-bit mode.

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 85 cd	JNE rel32	D	Valid	Valid	Jump near if not equal (ZF=0).
0F 8E cw	JNG rel16	D	N.S.	Valid	Jump near if not greater (ZF=1 or SF≠ OF). Not supported in 64-bit mode.
0F 8E cd	JNG rel32	D	Valid	Valid	Jump near if not greater (ZF=1 or SF≠ OF).
0F 8C cw	JNGE rel16	D	N.S.	Valid	Jump near if not greater or equal (SF≠ OF). Not supported in 64-bit mode.
0F 8C cd	JNGE rel32	D	Valid	Valid	Jump near if not greater or equal (SF≠ OF).
0F 8D cw	JNL rel16	D	N.S.	Valid	Jump near if not less (SF=OF). Not supported in 64-bit mode.
0F 8D cd	JNL rel32	D	Valid	Valid	Jump near if not less (SF=OF).
0F 8F cw	JNLE rel16	D	N.S.	Valid	Jump near if not less or equal (ZF=0 and SF=OF). Not supported in 64-bit mode.
0F 8F cd	JNLE rel32	D	Valid	Valid	Jump near if not less or equal (ZF=0 and SF=OF).
0F 81 cw	JNO rel16	D	N.S.	Valid	Jump near if not overflow (OF=0). Not supported in 64-bit mode.
0F 81 cd	JNO rel32	D	Valid	Valid	Jump near if not overflow (OF=0).
0F 8B cw	JNP rel16	D	N.S.	Valid	Jump near if not parity (PF=0). Not supported in 64-bit mode.
0F 8B cd	JNP rel32	D	Valid	Valid	Jump near if not parity (PF=0).
0F 89 cw	JNS rel16	D	N.S.	Valid	Jump near if not sign (SF=0). Not supported in 64-bit mode.
0F 89 cd	JNS rel32	D	Valid	Valid	Jump near if not sign (SF=0).
0F 85 cw	JNZ rel16	D	N.S.	Valid	Jump near if not zero (ZF=0). Not supported in 64-bit mode.
0F 85 cd	JNZ rel32	D	Valid	Valid	Jump near if not zero (ZF=0).
0F 80 cw	JO rel16	D	N.S.	Valid	Jump near if overflow (OF=1). Not supported in 64-bit mode.
0F 80 cd	JO rel32	D	Valid	Valid	Jump near if overflow (OF=1).
0F 8A cw	JP rel16	D	N.S.	Valid	Jump near if parity (PF=1). Not supported in 64-bit mode.
0F 8A cd	JP rel32	D	Valid	Valid	Jump near if parity (PF=1).
0F 8A cw	JPE rel16	D	N.S.	Valid	Jump near if parity even (PF=1). Not supported in 64-bit mode.
0F 8A cd	JPE rel32	D	Valid	Valid	Jump near if parity even (PF=1).
0F 8B cw	JPO rel16	D	N.S.	Valid	Jump near if parity odd (PF=0). Not supported in 64-bit mode.
0F 8B cd	JPO rel32	D	Valid	Valid	Jump near if parity odd (PF=0).
0F 88 cw	JS rel16	D	N.S.	Valid	Jump near if sign (SF=1). Not supported in 64-bit mode.
0F 88 cd	JS rel32	D	Valid	Valid	Jump near if sign (SF=1).
0F 84 cw	JZ rel16	D	N.S.	Valid	Jump near if 0 (ZF=1). Not supported in 64-bit mode.
0F 84 cd	JZ rel32	D	Valid	Valid	Jump near if 0 (ZF=1).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
D	Offset	N/A	N/A	N/A

Description

Checks the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and, if the flags are in the specified state (condition), performs a jump to the target instruction specified by the destination operand. A condition code (*cc*) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, the jump is not performed and execution continues with the instruction following the J*cc* instruction.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register). A relative offset (*rel8*, *rel16,* or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit or 32-bit immediate value, which is added to the instruction pointer. Instruction coding is most efficient for offsets of –128 to +127. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits.

The conditions for each J*cc* mnemonic are given in the "Description" column of the table on the preceding page. The terms "less" and "greater" are used for comparisons of signed integers and the terms "above" and "below" are used for unsigned integers.

Because a particular state of the status flags can sometimes be interpreted in two ways, two mnemonics are defined for some opcodes. For example, the JA (jump if above) instruction and the JNBE (jump if not below or equal) instruction are alternate mnemonics for the opcode 77H.

The J*cc* instruction does not support far jumps (jumps to other code segments). When the target for the conditional jump is in a different segment, use the opposite condition from the condition being tested for the J*cc* instruction, and then access the target with an unconditional far jump (JMP instruction) to the other segment. For example, the following conditional far jump is illegal:

```
JZ FARLABEL;
```

To accomplish this far jump, use the following two instructions:

```
JNZ BEYOND;
JMP FARLABEL;
BEYOND:
```

The JRCXZ, JECXZ, and JCXZ instructions differ from other J*cc* instructions because they do not check status flags. Instead, they check RCX, ECX or CX for 0. The register checked is determined by the address-size attribute. These instructions are useful when used at the beginning of a loop that terminates with a conditional loop instruction (such as LOOPNE). They can be used to prevent an instruction sequence from entering a loop when RCX, ECX or CX is 0. This would cause the loop to execute 2^{64}, 2^{32} or 64K times (not zero times).

All conditional jumps are converted to code fetches of one or two cache lines, regardless of jump address or cacheability.

In 64-bit mode, operand size is fixed at 64 bits. JMP Short is RIP = RIP + 8-bit offset sign extended to 64 bits. JMP Near is RIP = RIP + 32-bit offset sign extended to 64 bits.

Operation

```
IF condition
    THEN
        tempEIP := EIP + SignExtend(DEST);
        IF OperandSize = 16
            THEN tempEIP := tempEIP AND 0000FFFFH;
        FI;
    IF tempEIP is not within code segment limit
        THEN #GP(0);
        ELSE EIP := tempEIP
    FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the offset being jumped to is beyond the limits of the CS segment.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If the offset being jumped to is beyond the limits of the CS segment or is outside of the effective address space from 0 to FFFFH. This condition can occur if a 32-bit address size override prefix is used.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#UD	If the LOCK prefix is used.

JMP—Jump

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
EB cb	JMP rel8	D	Valid	Valid	Jump short, RIP = RIP + 8-bit displacement sign extended to 64-bits.
E9 cw	JMP rel16	D	N.S.	Valid	Jump near, relative, displacement relative to next instruction. Not supported in 64-bit mode.
E9 cd	JMP rel32	D	Valid	Valid	Jump near, relative, RIP = RIP + 32-bit displacement sign extended to 64-bits.
FF /4	JMP r/m16	M	N.S.	Valid	Jump near, absolute indirect, address = zero-extended r/m16. Not supported in 64-bit mode.
FF /4	JMP r/m32	M	N.S.	Valid	Jump near, absolute indirect, address given in r/m32. Not supported in 64-bit mode.
FF /4	JMP r/m64	M	Valid	N.E.	Jump near, absolute indirect, RIP = 64-Bit offset from register or memory.
EA cd	JMP ptr16:16	S	Inv.	Valid	Jump far, absolute, address given in operand.
EA cp	JMP ptr16:32	S	Inv.	Valid	Jump far, absolute, address given in operand.
FF /5	JMP m16:16	M	Valid	Valid	Jump far, absolute indirect, address given in m16:16.
FF /5	JMP m16:32	M	Valid	Valid	Jump far, absolute indirect, address given in m16:32.
REX.W FF /5	JMP m16:64	M	Valid	N.E.	Jump far, absolute indirect, address given in m16:64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
S	Segment + Absolute Address	N/A	N/A	N/A
D	Offset	N/A	N/A	N/A
M	ModRM:r/m (r)	N/A	N/A	N/A

Description

Transfers program control to a different point in the instruction stream without recording return information. The destination (target) operand specifies the address of the instruction being jumped to. This operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four different types of jumps:

- Near jump—A jump to an instruction within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment jump.
- Short jump—A near jump where the jump range is limited to –128 to +127 from the current EIP value.
- Far jump—A jump to an instruction located in a different segment than the current code segment but at the same privilege level, sometimes referred to as an intersegment jump.
- Task switch—A jump to an instruction located in a different task.

A task switch can only be executed in protected mode (see Chapter 10, in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, for information on performing task switches with the JMP instruction).

Near and Short Jumps. When executing a near jump, the processor jumps to the address (within the current code segment) that is specified with the target operand. The target operand specifies either an absolute offset (that is an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current

value of the instruction pointer in the EIP register). A near jump to a relative offset of 8-bits (*rel8*) is referred to as a short jump. The CS register is not changed on near and short jumps.

An absolute offset is specified indirectly in a general-purpose register or a memory location (*r/m16* or *r/m32*). The operand-size attribute determines the size of the target operand (16 or 32 bits). Absolute offsets are loaded directly into the EIP register. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits.

A relative offset (*rel8, rel16*, or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed 8-, 16-, or 32-bit immediate value. This value is added to the value in the EIP register. (Here, the EIP register contains the address of the instruction following the JMP instruction). When using relative offsets, the opcode (for short vs. near jumps) and the operand-size attribute (for near relative jumps) determines the size of the target operand (8, 16, or 32 bits).

Far Jumps in Real-Address or Virtual-8086 Mode. When executing a far jump in real-address or virtual-8086 mode, the processor jumps to the code segment and offset specified with the target operand. Here the target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). With the pointer method, the segment and address of the called procedure is encoded in the instruction, using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared.

Far Jumps in Protected Mode. When the processor is operating in protected mode, the JMP instruction can be used to perform the following three types of far jumps:

- A far jump to a conforming or non-conforming code segment.
- A far jump through a call gate.
- A task switch.

(The JMP instruction cannot be used to perform inter-privilege-level far jumps.)

In protected mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate, task gate, or TSS) and access rights determine the type of jump to be performed.

If the selected descriptor is for a code segment, a far jump to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far jump to the same privilege level in protected mode is very similar to one carried out in real-address or virtual-8086 mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register, and the offset from the instruction is loaded into the EIP register. Note that a call gate (described in the next paragraph) can also be used to perform far call to a code segment at the same privilege level. Using this mechanism provides an extra level of indirection and is the preferred method of making jumps between 16-bit and 32-bit code segments.

When executing a far jump through a call gate, the segment selector specified by the target operand identifies the call gate. (The offset part of the target operand is ignored.) The processor then jumps to the code segment specified in the call gate descriptor and begins executing the instruction at the offset specified in the call gate. No stack switch occurs. Here again, the target operand can specify the far address of the call gate either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*).

Executing a task switch with the JMP instruction is somewhat similar to executing a jump through a call gate. Here the target operand specifies the segment selector of the task gate for the task being switched to (and the offset part of the target operand is ignored). The task gate in turn points to the TSS for the task, which contains the segment selectors for the task's code and stack segments. The TSS also contains the EIP value for the next instruction that was to be executed before the task was suspended. This instruction pointer value is loaded into the EIP register so that the task begins executing again at this next instruction.

The JMP instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 10 in Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, for detailed information on the mechanics of a task switch.

When execution of a JMP instruction effects a task switch, the nested task flag (NT) is not set in the EFLAGS register and the new TSS's previous task link field is not loaded with the old task's TSS selector. A return to the previous task can thus not be carried out by executing the IRET instruction. Switching tasks with the JMP instruction differs in this regard from the CALL instruction which does set the NT flag and save the previous task link information, allowing a return to the calling task with an IRET instruction.

Refer to Chapter 6, "Procedure Calls, Interrupts, and Exceptions" and Chapter 18, "Control-flow Enforcement Technology (CET)" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for CET details.

In 64-Bit Mode. The instruction's operation size is fixed at 64 bits. If a selector points to a gate, then RIP equals the 64-bit displacement taken from gate; else RIP equals the zero-extended offset from the far pointer referenced in the instruction.

When FRED transitions are enabled, an execution of far JMP that references a call gate causes a general-protection exception, as does an execution of far JMP that would enter compatibility mode when CPL is 0.

See the summary chart at the beginning of this section for encoding data and limits.

Instruction ordering. Instructions following a far jump may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the far jump have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

Instructions sequentially following a near indirect JMP instruction (i.e., those not at the target) may be executed speculatively. If software needs to prevent this (e.g., in order to prevent a speculative execution side channel), then an INT3 or LFENCE instruction opcode can be placed after the near indirect JMP in order to block speculative execution.

Operation

```
IF near jump
    IF 64-bit Mode
        THEN
            IF near relative jump
             THEN
                tempRIP := RIP + DEST; (* RIP is instruction following JMP instruction*)
             ELSE (* Near absolute jump *)
                tempRIP := DEST;
            FI;
        ELSE
            IF near relative jump
             THEN
                tempEIP := EIP + DEST; (* EIP is instruction following JMP instruction*)
             ELSE (* Near absolute jump *)
                tempEIP := DEST;
            FI;
    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 OperandSize = 32
         THEN
            EIP := tempEIP;
         ELSE
            IF OperandSize = 16
                THEN (* OperandSize = 16 *)
                    EIP := tempEIP AND 0000FFFFH;
                ELSE (* OperandSize = 64)
```

```
                    RIP := tempRIP;
                FI;
        FI;
    IF (JMP near indirect, absolute indirect)
        IF EndbranchEnabledAndNotSuppressed(CPL)
            IF CPL = 3
                THEN
                    IF ( no 3EH prefix OR IA32_U_CET.NO_TRACK_EN == 0 )
                        THEN
                            IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
                    FI;
                ELSE
                    IF ( no 3EH prefix OR IA32_S_CET.NO_TRACK_EN == 0 )
                        THEN
                            IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                    FI;
            FI;
        FI;
    FI;
FI;
IF far jump and (PE = 0 or (PE = 1 AND VM = 1)) (* Real-address or virtual-8086 mode *)
    THEN
        tempEIP := DEST(Offset); (* DEST is ptr16:32 or [m16:32] *)
        IF tempEIP is beyond code segment limit
            THEN #GP(0); FI;
        CS := DEST(segment selector); (* DEST is ptr16:32 or [m16:32] *)
        IF OperandSize = 32
            THEN
                EIP := tempEIP; (* DEST is ptr16:32 or [m16:32] *)
            ELSE (* OperandSize = 16 *)
                EIP := tempEIP AND 0000FFFFH; (* Clear upper 16 bits *)
        FI;
FI;
IF far jump and (PE = 1 and VM = 0)
(* IA-32e mode or protected mode, not virtual-8086 mode *)
    THEN
        IF effective address in the CS, DS, ES, FS, GS, or SS segment is illegal or segment selector in target operand NULL
                THEN #GP(0); FI;
        IF segment selector index not within descriptor table limits
            THEN #GP(new selector); FI;
        Read type and access rights of segment descriptor;
        IF (IA32_EFER.LMA = 0)
            THEN
                IF segment type is not a conforming or nonconforming code segment, call gate, task gate, or TSS
                    THEN #GP(segment selector); FI;
            ELSE
                IF segment type is not a conforming or nonconforming code segment or call gate
                    THEN #GP(segment selector); FI;
        FI;
        Depending on type and access rights:
            GO TO CONFORMING-CODE-SEGMENT;
            GO TO NONCONFORMING-CODE-SEGMENT;
            GO TO CALL-GATE;
            GO TO TASK-GATE;
```

```
                GO TO TASK-STATE-SEGMENT;
        ELSE
            #GP(segment selector);
FI;
CONFORMING-CODE-SEGMENT:
    IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
        THEN GP(new code segment selector); FI;
    IF DPL > CPL
        THEN #GP(segment selector); FI;
    IF CR4.FRED = 1 and CPL = 0 and L-bit = 0
        THEN GP(new code segment selector); FI;
    IF segment not present
        THEN #NP(segment selector); FI;
    tempEIP := DEST(Offset);
    IF OperandSize = 16
        THEN tempEIP := tempEIP AND 0000FFFFH;
    FI;
    IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode) and
    tempEIP outside code segment limit
        THEN #GP(0); FI
    IF tempEIP is non-canonical
        THEN #GP(0); FI;
    IF ShadowStackEnabled(CPL)
        IF (IA32_EFER.LMA and DEST(segment selector).L) = 0
            (* If target is legacy or compatibility mode then the SSP must be in low 4GB *)
            IF (SSP & 0xFFFFFFFF00000000 != 0)
                THEN #GP(0); FI;
        FI;
    FI;
    CS := DEST[segment selector]; (* Segment descriptor information also loaded *)
    CS(RPL) := CPL
    EIP := tempEIP;
    IF EndbranchEnabled(CPL)
        IF CPL = 3
            THEN
                IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
                IA32_U_CET.SUPPRESS = 0
            ELSE
                IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                IA32_S_CET.SUPPRESS = 0
        FI;
    FI;
END;
NONCONFORMING-CODE-SEGMENT:
    IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
        THEN GP(new code segment selector); FI;
    IF (RPL > CPL) OR (DPL ≠ CPL)
        THEN #GP(code segment selector); FI;
    IF CR4.FRED = 1 and CPL = 0 and L-bit = 0
        THEN GP(new code segment selector); FI;
    IF segment not present
        THEN #NP(segment selector); FI;
    tempEIP := DEST(Offset);
    IF OperandSize = 16
```

```
        THEN tempEIP := tempEIP AND 0000FFFFH; FI;
    IF (IA32_EFER.LMA = 0 OR target mode = Compatibility mode)
    and tempEIP outside code segment limit
        THEN #GP(0); FI
    IF tempEIP is non-canonical THEN #GP(0); FI;
    IF ShadowStackEnabled(CPL)
        IF (IA32_EFER.LMA and DEST(segment selector).L) = 0
            (* If target is legacy or compatibility mode then the SSP must be in low 4GB *)
            IF (SSP & 0xFFFFFFFF00000000 != 0)
                THEN #GP(0); FI;
        FI;
    FI;
    CS := DEST[segment selector]; (* Segment descriptor information also loaded *)
    CS(RPL) := CPL;
    EIP := tempEIP;
    IF EndbranchEnabled(CPL)
        IF CPL = 3
            THEN
                IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
                IA32_U_CET.SUPPRESS = 0
            ELSE
                IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                IA32_S_CET.SUPPRESS = 0
        FI;
    FI;
END;

CALL-GATE:
    IF call gate DPL < CPL or call gate DPL < call gate segment-selector RPL or CR4.FRED = 1
            THEN #GP(call gate selector); FI;
    IF call gate not present
        THEN #NP(call gate selector); FI;
    IF call gate code-segment selector is NULL
        THEN #GP(0); FI;
    IF call gate code-segment selector index outside descriptor table limits
        THEN #GP(code segment selector); FI;
    Read code segment descriptor;
    IF code-segment segment descriptor does not indicate a code segment
    or code-segment segment descriptor is conforming and DPL > CPL
    or code-segment segment descriptor is non-conforming and DPL ≠ CPL
            THEN #GP(code segment selector); FI;
    IF IA32_EFER.LMA = 1 and (code-segment descriptor is not a 64-bit code segment
    or code-segment segment descriptor has both L-Bit and D-bit set)
            THEN #GP(code segment selector); FI;
    IF code segment is not present
        THEN #NP(code-segment selector); FI;
     tempEIP := DEST(Offset);
     IF GateSize = 16
        THEN tempEIP := tempEIP AND 0000FFFFH; FI;
    IF (IA32_EFER.LMA = 0 OR target mode = Compatibility mode) AND tempEIP
    outside code segment limit
        THEN #GP(0); FI
    CS := DEST[SegmentSelector]; (* Segment descriptor information also loaded *)
    CS(RPL) := CPL;
```

```
    EIP := tempEIP;
    IF EndbranchEnabled(CPL)
        IF CPL = 3
            THEN
                IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH;
                IA32_U_CET.SUPPRESS = 0
            ELSE
                IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH;
                IA32_S_CET.SUPPRESS = 0
        FI;
    FI;
END;
TASK-GATE:
    IF task gate DPL < CPL
    or task gate DPL < task gate segment-selector RPL
        THEN #GP(task gate selector); FI;
    IF task gate not present
        THEN #NP(gate selector); FI;
    Read the TSS segment selector in the task-gate descriptor;
    IF TSS segment selector local/global bit is set to local
    or index not within GDT limits
    or descriptor is not a TSS segment
    or TSS descriptor specifies that the TSS is busy
        THEN #GP(TSS selector); FI;
     IF TSS not present
        THEN #NP(TSS selector); FI;
     SWITCH-TASKS to TSS;
     IF EIP not within code segment limit
        THEN #GP(0); FI;
END;
TASK-STATE-SEGMENT:
    IF TSS DPL < CPL
    or TSS DPL < TSS segment-selector RPL
    or TSS descriptor indicates TSS not available
        THEN #GP(TSS selector); FI;
    IF TSS is not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;
```

Flags Affected

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

Protected Mode Exceptions

#GP(0)	If offset in target operand, call gate, or TSS is beyond the code segment limits.
	If the segment selector in the destination operand, call gate, task gate, or TSS is NULL.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If target mode is compatibility mode and SSP is not in low 4GB.

#GP(selector)	If the segment selector index is outside descriptor table limits.
	If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment.
	If the DPL for a nonconforming-code segment is not equal to the CPL
	(When not using a call gate.) If the RPL for the segment's segment selector is greater than the CPL.
	If the DPL for a conforming-code segment is greater than the CPL.
	If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS's segment selector.
	If the segment descriptor for selector in a call gate does not indicate it is a code segment.
	If the segment descriptor for the segment selector in a task gate does not indicate an available TSS.
	If the segment selector for a TSS has its local/global bit set for local.
	If a TSS segment descriptor specifies that the TSS is busy or not available.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NP (selector)	If the code segment being accessed is not present.
	If call gate, task gate, or TSS not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. (Only occurs when fetching target from memory.)
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same as 64-bit mode exceptions.

#GP(Selector)	If FRED transitions are enabled and the selector references a call gate.

64-Bit Mode Exceptions

#GP(0)	If a memory address is non-canonical.
	If target offset in destination operand is non-canonical.
	If target offset in destination operand is beyond the new code segment limit.
	If the segment selector in the destination operand is NULL.
	If the code segment selector in the 64-bit gate is NULL.
	If transitioning to compatibility mode and the SSP is beyond 4GB.

#GP(selector)	If the code segment or 64-bit call gate is outside descriptor table limits.
	If the code segment or 64-bit call gate overlaps non-canonical space.
	If the segment descriptor from a 64-bit call gate is in non-canonical space.
	If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, 64-bit call gate.
	If the segment descriptor pointed to by the segment selector in the destination operand is a code segment, and has both the D-bit and the L-bit set.
	If the DPL for a nonconforming-code segment is not equal to the CPL, or the RPL for the segment's segment selector is greater than the CPL.
	If the DPL for a conforming-code segment is greater than the CPL.
	If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate.
	If the upper type field of a 64-bit call gate is not 0x0.
	If the segment selector from a 64-bit call gate is beyond the descriptor table limits.
	If the code segment descriptor pointed to by the selector in the 64-bit gate doesn't have the L-bit set and the D-bit clear.
	If the segment descriptor for a segment selector from the 64-bit call gate does not indicate it is a code segment.
	If the code segment is non-conforming and CPL ≠ DPL.
	If the code segment is confirming and CPL < DPL.
	If FRED transitions are enabled and the selector references a call gate.
	If FRED transitions are enabled, CPL is 0, and the instruction would enter compatibility mode.
#NP(selector)	If a code segment or 64-bit call gate is not present.
#UD	(64-bit mode only) If a far jump is direct to an absolute address in memory.
	If the LOCK prefix is used.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

KADDW/KADDB/KADDQ/KADDD—ADD Two Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L1.0F.W0 4A /r KADDW k1, k2, k3	RVR	V/V	AVX512DQ OR AVX10.1	Add 16 bits masks in k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 4A /r KADDB k1, k2, k3	RVR	V/V	AVX512DQ OR AVX10.1	Add 8 bits masks in k2 and k3 and place result in k1.
VEX.L1.0F.W1 4A /r KADDQ k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1	Add 64 bits masks in k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 4A /r KADDD k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1	Add 32 bits masks in k2 and k3 and place result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Adds the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

Operation

KADDW

```
DEST[15:0] := SRC1[15:0] + SRC2[15:0]
DEST[MAX_KL-1:16] := 0
```

KADDB

```
DEST[7:0] := SRC1[7:0] + SRC2[7:0]
DEST[MAX_KL-1:8] := 0
```

KADDQ

```
DEST[63:0] := SRC1[63:0] + SRC2[63:0]
DEST[MAX_KL-1:64] := 0
```

KADDD

```
DEST[31:0] := SRC1[31:0] + SRC2[31:0]
DEST[MAX_KL-1:32] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

KADDW __mmask16 _kadd_mask16 (__mmask16 a, __mmask16 b);
KADDB __mmask8 _kadd_mask8 (__mmask8 a, __mmask8 b);
KADDQ __mmask64 _kadd_mask64 (__mmask64 a, __mmask64 b);
KADDD __mmask32 _kadd_mask32 (__mmask32 a, __mmask32 b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-65, "TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)."

KANDNW/KANDNB/KANDNQ/KANDND—Bitwise Logical AND NOT Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L1.0F.W0 42 /r KANDNW k1, k2, k3	RVR	V/V	AVX512F OR AVX10.1	Bitwise AND NOT 16 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 42 /r KANDNB k1, k2, k3	RVR	V/V	AVX512DQ OR AVX10.1	Bitwise AND NOT 8 bits masks k1 and k2 and place result in k1.
VEX.L1.0F.W1 42 /r KANDNQ k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1	Bitwise AND NOT 64 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 42 /r KANDND k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1	Bitwise AND NOT 32 bits masks k2 and k3 and place result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise AND NOT between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

Operation

KANDNW

```
DEST[15:0] := (BITWISE NOT SRC1[15:0]) BITWISE AND SRC2[15:0]
DEST[MAX_KL-1:16] := 0
```

KANDNB

```
DEST[7:0] := (BITWISE NOT SRC1[7:0]) BITWISE AND SRC2[7:0]
DEST[MAX_KL-1:8] := 0
```

KANDNQ

```
DEST[63:0] := (BITWISE NOT SRC1[63:0]) BITWISE AND SRC2[63:0]
DEST[MAX_KL-1:64] := 0
```

KANDND

```
DEST[31:0] := (BITWISE NOT SRC1[31:0]) BITWISE AND SRC2[31:0]
DEST[MAX_KL-1:32] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

KANDNW __mmask16 _mm512_kandn(__mmask16 a, __mmask16 b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-65, "TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)."

KANDW/KANDB/KANDQ/KANDD—Bitwise Logical AND Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L1.0F.W0 41 /r KANDW k1, k2, k3	RVR	V/V	AVX512F OR AVX10.1	Bitwise AND 16 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 41 /r KANDB k1, k2, k3	RVR	V/V	AVX512DQ OR AVX10.1	Bitwise AND 8 bits masks k2 and k3 and place result in k1.
VEX.L1.0F.W1 41 /r KANDQ k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1	Bitwise AND 64 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 41 /r KANDD k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1	Bitwise AND 32 bits masks k2 and k3 and place result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise AND between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

Operation

KANDW

```
DEST[15:0] := SRC1[15:0] BITWISE AND SRC2[15:0]
DEST[MAX_KL-1:16] := 0
```

KANDB

```
DEST[7:0] := SRC1[7:0] BITWISE AND SRC2[7:0]
DEST[MAX_KL-1:8] := 0
```

KANDQ

```
DEST[63:0] := SRC1[63:0] BITWISE AND SRC2[63:0]
DEST[MAX_KL-1:64] := 0
```

KANDD

```
DEST[31:0] := SRC1[31:0] BITWISE AND SRC2[31:0]
DEST[MAX_KL-1:32] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

KANDW __mmask16 _mm512_kand(__mmask16 a, __mmask16 b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-65, "TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)."

KMOVW/KMOVB/KMOVQ/KMOVD—Move From and to Mask Registers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.0F.W0 90 /r KMOVW k1, k2/m16	RM	V/V	AVX512F OR AVX10.1	Move 16 bits mask from k2/m16 and store the result in k1.
VEX.L0.66.0F.W0 90 /r KMOVB k1, k2/m8	RM	V/V	AVX512DQ OR AVX10.1	Move 8 bits mask from k2/m8 and store the result in k1.
VEX.L0.0F.W1 90 /r KMOVQ k1, k2/m64	RM	V/V	AVX512BW OR AVX10.1	Move 64 bits mask from k2/m64 and store the result in k1.
VEX.L0.66.0F.W1 90 /r KMOVD k1, k2/m32	RM	V/V	AVX512BW OR AVX10.1	Move 32 bits mask from k2/m32 and store the result in k1.
VEX.L0.0F.W0 91 /r KMOVW m16, k1	MR	V/V	AVX512F OR AVX10.1	Move 16 bits mask from k1 and store the result in m16.
VEX.L0.66.0F.W0 91 /r KMOVB m8, k1	MR	V/V	AVX512DQ OR AVX10.1	Move 8 bits mask from k1 and store the result in m8.
VEX.L0.0F.W1 91 /r KMOVQ m64, k1	MR	V/V	AVX512BW OR AVX10.1	Move 64 bits mask from k1 and store the result in m64.
VEX.L0.66.0F.W1 91 /r KMOVD m32, k1	MR	V/V	AVX512BW OR AVX10.1	Move 32 bits mask from k1 and store the result in m32.
VEX.L0.0F.W0 92 /r KMOVW k1, r32	RR	V/V	AVX512F OR AVX10.1	Move 16 bits mask from r32 to k1.
VEX.L0.66.0F.W0 92 /r KMOVB k1, r32	RR	V/V	AVX512DQ OR AVX10.1	Move 8 bits mask from r32 to k1.
VEX.L0.F2.0F.W1 92 /r KMOVQ k1, r64	RR	V/I	AVX512BW OR AVX10.1	Move 64 bits mask from r64 to k1.
VEX.L0.F2.0F.W0 92 /r KMOVD k1, r32	RR	V/V	AVX512BW OR AVX10.1	Move 32 bits mask from r32 to k1.
VEX.L0.0F.W0 93 /r KMOVW r32, k1	RR	V/V	AVX512F OR AVX10.1	Move 16 bits mask from k1 to r32.
VEX.L0.66.0F.W0 93 /r KMOVB r32, k1	RR	V/V	AVX512DQ OR AVX10.1	Move 8 bits mask from k1 to r32.
VEX.L0.F2.0F.W1 93 /r KMOVQ r64, k1	RR	V/I	AVX512BW OR AVX10.1	Move 64 bits mask from k1 to r64.
VEX.L0.F2.0F.W0 93 /r KMOVD r32, k1	RR	V/V	AVX512BW OR AVX10.1	Move 32 bits mask from k1 to r32.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2
RM	ModRM:reg (w)	ModRM:r/m (r)
MR	ModRM:r/m (w, ModRM:[7:6] must not be 11b)	ModRM:reg (r)
RR	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Copies values from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be mask registers, memory location or general purpose. The instruction cannot be used to transfer data between general purpose registers and or memory locations.

When moving to a mask register, the result is zero extended to MAX_KL size (i.e., 64 bits currently). When moving to a general-purpose register (GPR), the result is zero-extended to the size of the destination. In 32-bit mode, the default GPR destination's size is 32 bits. In 64-bit mode, the default GPR destination's size is 64 bits. Note that VEX.W can only be used to modify the size of the GPR operand in 64b mode.

Operation

KMOVW

```
IF *destination is a memory location*
    DEST[15:0] := SRC[15:0]
IF *destination is a mask register or a GPR *
    DEST := ZeroExtension(SRC[15:0])
```

KMOVB

```
IF *destination is a memory location*
    DEST[7:0] := SRC[7:0]
IF *destination is a mask register or a GPR *
    DEST := ZeroExtension(SRC[7:0])
```

KMOVQ

```
IF *destination is a memory location or a GPR*
    DEST[63:0] := SRC[63:0]
IF *destination is a mask register*
    DEST := ZeroExtension(SRC[63:0])
```

KMOVD

```
IF *destination is a memory location*
    DEST[31:0] := SRC[31:0]
IF *destination is a mask register or a GPR *
    DEST := ZeroExtension(SRC[31:0])
```

Intel C/C++ Compiler Intrinsic Equivalent

KMOVW __mmask16 _mm512_kmov(__mmask16 a);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Instructions with RR operand encoding, see Table 2-65, "TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)."

Instructions with RM or MR operand encoding, see Table 2-66, "TYPE K21 Exception Definition (VEX-Encoded OpMask Instructions Addressing Memory)."

KNOTW/KNOTB/KNOTQ/KNOTD—NOT Mask Register

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.0F.W0 44 /r KNOTW k1, k2	RR	V/V	AVX512F OR AVX10.1	Bitwise NOT of 16 bits mask k2.
VEX.L0.66.0F.W0 44 /r KNOTB k1, k2	RR	V/V	AVX512DQ OR AVX10.1	Bitwise NOT of 8 bits mask k2.
VEX.L0.0F.W1 44 /r KNOTQ k1, k2	RR	V/V	AVX512BW OR AVX10.1	Bitwise NOT of 64 bits mask k2.
VEX.L0.66.0F.W1 44 /r KNOTD k1, k2	RR	V/V	AVX512BW OR AVX10.1	Bitwise NOT of 32 bits mask k2.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2
RR	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise NOT of vector mask k2 and writes the result into vector mask k1.

Operation

KNOTW

```
DEST[15:0] := BITWISE NOT SRC[15:0]
DEST[MAX_KL-1:16] := 0
```

KNOTB

```
DEST[7:0] := BITWISE NOT SRC[7:0]
DEST[MAX_KL-1:8] := 0
```

KNOTQ

```
DEST[63:0] := BITWISE NOT SRC[63:0]
DEST[MAX_KL-1:64] := 0
```

KNOTD

```
DEST[31:0] := BITWISE NOT SRC[31:0]
DEST[MAX_KL-1:32] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

KNOTW __mmask16 _mm512_knot(__mmask16 a);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-65, "TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)."

KORTESTW/KORTESTB/KORTESTQ/KORTESTD—OR Masks and Set Flags

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.0F.W0 98 /r KORTESTW k1, k2	RR	V/V	AVX512F OR AVX10.1	Bitwise OR 16 bits masks k1 and k2 and update ZF and CF accordingly.
VEX.L0.66.0F.W0 98 /r KORTESTB k1, k2	RR	V/V	AVX512DQ OR AVX10.1	Bitwise OR 8 bits masks k1 and k2 and update ZF and CF accordingly.
VEX.L0.0F.W1 98 /r KORTESTQ k1, k2	RR	V/V	AVX512BW OR AVX10.1	Bitwise OR 64 bits masks k1 and k2 and update ZF and CF accordingly.
VEX.L0.66.0F.W1 98 /r KORTESTD k1, k2	RR	V/V	AVX512BW OR AVX10.1	Bitwise OR 32 bits masks k1 and k2 and update ZF and CF accordingly.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2
RR	ModRM:reg (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise OR between the vector mask register k2, and the vector mask register k1, and sets CF and ZF based on the operation result.

ZF flag is set if both sources are 0x0. CF is set if, after the OR operation is done, the operation result is all 1's.

Operation

KORTESTW

```
TMP[15:0] := SRC2[15:0] BITWISE OR SRC1[15:0]
IF(TMP[15:0]=0)
    THEN ZF := 1
    ELSE ZF := 0
FI;
IF(TMP[15:0]=FFFFH)
    THEN CF := 1
    ELSE CF := 0
FI;
AF := OF := PF := SF := 0;
```

KORTESTB

```
TMP[7:0] := SRC2[7:0] BITWISE OR SRC1[7:0]
IF(TMP[7:0]=0)
    THEN ZF := 1
    ELSE ZF := 0
FI;
IF(TMP[7:0]==FFH)
    THEN CF := 1
    ELSE CF := 0
FI;
AF := OF := PF := SF := 0;
```

KORTESTQ

```
TMP[63:0] := SRC2[63:0] BITWISE OR SRC1[63:0]
IF(TMP[63:0]=0)
    THEN ZF := 1
    ELSE ZF := 0
FI;
IF(TMP[63:0]==FFFFFFFF_FFFFFFFFH)
    THEN CF := 1
    ELSE CF := 0
FI;
AF := OF := PF := SF := 0;
```

KORTESTD

```
TMP[31:0] := SRC2[31:0] BITWISE OR SRC1[31:0]
IF(TMP[31:0]=0)
    THEN ZF := 1
    ELSE ZF := 0
FI;
IF(TMP[31:0]=FFFFFFFFH)
    THEN CF := 1
    ELSE CF := 0
FI;
AF := OF := PF := SF := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

KORTESTW __mmask16 _mm512_kortest[cz](__mmask16 a, __mmask16 b);

Flags Affected

The ZF flag is set if the result of OR-ing both sources is all 0s.

The CF flag is set if the result of OR-ing both sources is all 1s.

The OF, SF, AF, and PF flags are set to 0.

Other Exceptions

See Table 2-65, "TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)."

KORW/KORB/KORQ/KORD—Bitwise Logical OR Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L1.0F.W0 45 /r KORW k1, k2, k3	RVR	V/V	AVX512F OR AVX10.1	Bitwise OR 16 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 45 /r KORB k1, k2, k3	RVR	V/V	AVX512DQ OR AVX10.1	Bitwise OR 8 bits masks k2 and k3 and place result in k1.
VEX.L1.0F.W1 45 /r KORQ k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1	Bitwise OR 64 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 45 /r KORD k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1	Bitwise OR 32 bits masks k2 and k3 and place result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise OR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

Operation

KORW

```
DEST[15:0] := SRC1[15:0] BITWISE OR SRC2[15:0]
DEST[MAX_KL-1:16] := 0
```

KORB

```
DEST[7:0] := SRC1[7:0] BITWISE OR SRC2[7:0]
DEST[MAX_KL-1:8] := 0
```

KORQ

```
DEST[63:0] := SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[MAX_KL-1:64] := 0
```

KORD

```
DEST[31:0] := SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[MAX_KL-1:32] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

KORW __mmask16 _mm512_kor(__mmask16 a, __mmask16 b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-65, "TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)."

KSHIFTLW/KSHIFTLB/KSHIFTLQ/KSHIFTLD—Shift Left Mask Registers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.66.0F3A.W1 32 /r KSHIFTLW k1, k2, imm8	RRI	V/V	AVX512F OR AVX10.1	Shift left 16 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W0 32 /r KSHIFTLB k1, k2, imm8	RRI	V/V	AVX512DQ OR AVX10.1	Shift left 8 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W1 33 /r KSHIFTLQ k1, k2, imm8	RRI	V/V	AVX512BW OR AVX10.1	Shift left 64 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W0 33 /r KSHIFTLD k1, k2, imm8	RRI	V/V	AVX512BW OR AVX10.1	Shift left 32 bits in k2 by immediate and write result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RRI	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)	imm8

Description

Shifts 8/16/32/64 bits in the second operand (source operand) left by the count specified in immediate byte and place the least significant 8/16/32/64 bits of the result in the destination operand. The higher bits of the destination are zero-extended. The destination is set to zero if the count value is greater than 7 (for byte shift), 15 (for word shift), 31 (for doubleword shift) or 63 (for quadword shift).

Operation

KSHIFTLW

```
COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <=15
    THEN DEST[15:0] := SRC1[15:0] << COUNT;
FI;
```

KSHIFTLB

```
COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <=7
        THEN    DEST[7:0] := SRC1[7:0] << COUNT;
FI;
```

KSHIFTLQ

```
COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <=63
        THEN    DEST[63:0] := SRC1[63:0] << COUNT;
FI;
```

KSHIFTLD

```
COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <=31
        THEN    DEST[31:0] := SRC1[31:0] << COUNT;
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

Compiler auto generates KSHIFTLW when needed.

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-65, "TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)."

KSHIFTRW/KSHIFTRB/KSHIFTRQ/KSHIFTRD—Shift Right Mask Registers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.66.0F3A.W1 30 /r KSHIFTRW k1, k2, imm8	RRI	V/V	AVX512F OR AVX10.1	Shift right 16 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W0 30 /r KSHIFTRB k1, k2, imm8	RRI	V/V	AVX512DQ OR AVX10.1	Shift right 8 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W1 31 /r KSHIFTRQ k1, k2, imm8	RRI	V/V	AVX512BW OR AVX10.1	Shift right 64 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W0 31 /r KSHIFTRD k1, k2, imm8	RRI	V/V	AVX512BW OR AVX10.1	Shift right 32 bits in k2 by immediate and write result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RRI	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)	imm8

Description

Shifts 8/16/32/64 bits in the second operand (source operand) right by the count specified in immediate and place the least significant 8/16/32/64 bits of the result in the destination operand. The higher bits of the destination are zero-extended. The destination is set to zero if the count value is greater than 7 (for byte shift), 15 (for word shift), 31 (for doubleword shift) or 63 (for quadword shift).

Operation

KSHIFTRW

```
COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <=15
    THEN DEST[15:0] := SRC1[15:0] >> COUNT;
FI;
```

KSHIFTRB

```
COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <=7
        THEN    DEST[7:0] := SRC1[7:0] >> COUNT;
FI;
```

KSHIFTRQ

```
COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <=63
        THEN    DEST[63:0] := SRC1[63:0] >> COUNT;
FI;
```

KSHIFTRD

```
COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <=31
        THEN    DEST[31:0] := SRC1[31:0] >> COUNT;
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

Compiler auto generates KSHIFTRW when needed.

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-65, "TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)."

KTESTW/KTESTB/KTESTQ/KTESTD—Packed Bit Test Masks and Set Flags

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.0F.W0 99 /r KTESTW k1, k2	RR	V/V	AVX512DQ OR AVX10.1	Set ZF and CF depending on sign bit AND and ANDN of 16 bits mask register sources.
VEX.L0.66.0F.W0 99 /r KTESTB k1, k2	RR	V/V	AVX512DQ OR AVX10.1	Set ZF and CF depending on sign bit AND and ANDN of 8 bits mask register sources.
VEX.L0.0F.W1 99 /r KTESTQ k1, k2	RR	V/V	AVX512BW OR AVX10.1	Set ZF and CF depending on sign bit AND and ANDN of 64 bits mask register sources.
VEX.L0.66.0F.W1 99 /r KTESTD k1, k2	RR	V/V	AVX512BW OR AVX10.1	Set ZF and CF depending on sign bit AND and ANDN of 32 bits mask register sources.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2
RR	ModRM:reg (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise comparison of the bits of the first source operand and corresponding bits in the second source operand. If the AND operation produces all zeros, the ZF is set else the ZF is clear. If the bitwise AND operation of the inverted first source operand with the second source operand produces all zeros the CF is set else the CF is clear. Only the EFLAGS register is updated.

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

Operation

KTESTW

```
TEMP[15:0] := SRC2[15:0] AND SRC1[15:0]
IF (TEMP[15:0] = = 0)
    THEN ZF :=1;
    ELSE ZF := 0;
FI;
TEMP[15:0] := SRC2[15:0] AND NOT SRC1[15:0]
IF (TEMP[15:0] = = 0)
    THEN CF :=1;
    ELSE CF := 0;
FI;
AF := OF := PF := SF := 0;
```

KTESTB

```
TEMP[7:0] := SRC2[7:0] AND SRC1[7:0]
IF (TEMP[7:0] = = 0)
    THEN ZF :=1;
    ELSE ZF := 0;
FI;
TEMP[7:0] := SRC2[7:0] AND NOT SRC1[7:0]
IF (TEMP[7:0] = = 0)
    THEN CF :=1;
    ELSE CF := 0;
FI;
AF := OF := PF := SF := 0;
```

KTESTQ

```
TEMP[63:0] := SRC2[63:0] AND SRC1[63:0]
IF (TEMP[63:0] = = 0)
    THEN ZF :=1;
    ELSE ZF := 0;
FI;
TEMP[63:0] := SRC2[63:0] AND NOT SRC1[63:0]
IF (TEMP[63:0] = = 0)
    THEN CF :=1;
    ELSE CF := 0;
FI;
AF := OF := PF := SF := 0;
```

KTESTD

```
TEMP[31:0] := SRC2[31:0] AND SRC1[31:0]
IF (TEMP[31:0] = = 0)
    THEN ZF :=1;
    ELSE ZF := 0;
FI;
TEMP[31:0] := SRC2[31:0] AND NOT SRC1[31:0]
IF (TEMP[31:0] = = 0)
    THEN CF :=1;
    ELSE CF := 0;
FI;
AF := OF := PF := SF := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-65, "TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)."

KUNPCKBW/KUNPCKWD/KUNPCKDQ—Unpack for Mask Registers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L1.66.0F.W0 4B /r KUNPCKBW k1, k2, k3	RVR	V/V	AVX512F OR AVX10.1	Unpack 8-bit masks in k2 and k3 and write word result in k1.
VEX.L1.0F.W0 4B /r KUNPCKWD k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1	Unpack 16-bit masks in k2 and k3 and write doubleword result in k1.
VEX.L1.0F.W1 4B /r KUNPCKDQ k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1	Unpack 32-bit masks in k2 and k3 and write quadword result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Unpacks the lower 8/16/32 bits of the second and third operands (source operands) into the low part of the first operand (destination operand), starting from the low bytes. The result is zero-extended in the destination.

Operation

KUNPCKBW

```
DEST[7:0] := SRC2[7:0]
DEST[15:8] := SRC1[7:0]
DEST[MAX_KL-1:16] := 0
```

KUNPCKWD

```
DEST[15:0] := SRC2[15:0]
DEST[31:16] := SRC1[15:0]
DEST[MAX_KL-1:32] := 0
```

KUNPCKDQ

```
DEST[31:0] := SRC2[31:0]
DEST[63:32] := SRC1[31:0]
DEST[MAX_KL-1:64] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

KUNPCKBW __mmask16 _mm512_kunpackb(__mmask16 a, __mmask16 b);
KUNPCKDQ __mmask64 _mm512_kunpackd(__mmask64 a, __mmask64 b);
KUNPCKWD __mmask32 _mm512_kunpackw(__mmask32 a, __mmask32 b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-65, "TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)."

KXNORW/KXNORB/KXNORQ/KXNORD—Bitwise Logical XNOR Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L1.0F.W0 46 /r KXNORW k1, k2, k3	RVR	V/V	AVX512F OR AVX10.1	Bitwise XNOR 16-bit masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 46 /r KXNORB k1, k2, k3	RVR	V/V	AVX512DQ OR AVX10.1	Bitwise XNOR 8-bit masks k2 and k3 and place result in k1.
VEX.L1.0F.W1 46 /r KXNORQ k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1	Bitwise XNOR 64-bit masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 46 /r KXNORD k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1	Bitwise XNOR 32-bit masks k2 and k3 and place result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise XNOR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

Operation

KXNORW

```
DEST[15:0] := NOT (SRC1[15:0] BITWISE XOR SRC2[15:0])
DEST[MAX_KL-1:16] := 0
```

KXNORB

```
DEST[7:0] := NOT (SRC1[7:0] BITWISE XOR SRC2[7:0])
DEST[MAX_KL-1:8] := 0
```

KXNORQ

```
DEST[63:0] := NOT (SRC1[63:0] BITWISE XOR SRC2[63:0])
DEST[MAX_KL-1:64] := 0
```

KXNORD

```
DEST[31:0] := NOT (SRC1[31:0] BITWISE XOR SRC2[31:0])
DEST[MAX_KL-1:32] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

KXNORW __mmask16 _mm512_kxnor(__mmask16 a, __mmask16 b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-65, "TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)."

KXORW/KXORB/KXORQ/KXORD—Bitwise Logical XOR Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L1.0F.W0 47 /r KXORW k1, k2, k3	RVR	V/V	AVX512F OR AVX10.1	Bitwise XOR 16-bit masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 47 /r KXORB k1, k2, k3	RVR	V/V	AVX512DQ OR AVX10.1	Bitwise XOR 8-bit masks k2 and k3 and place result in k1.
VEX.L1.0F.W1 47 /r KXORQ k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1	Bitwise XOR 64-bit masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 47 /r KXORD k1, k2, k3	RVR	V/V	AVX512BW OR AVX10.1	Bitwise XOR 32-bit masks k2 and k3 and place result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise XOR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

Operation

KXORW

```
DEST[15:0] := SRC1[15:0] BITWISE XOR SRC2[15:0]
DEST[MAX_KL-1:16] := 0
```

KXORB

```
DEST[7:0] := SRC1[7:0] BITWISE XOR SRC2[7:0]
DEST[MAX_KL-1:8] := 0
```

KXORQ

```
DEST[63:0] := SRC1[63:0] BITWISE XOR SRC2[63:0]
DEST[MAX_KL-1:64] := 0
```

KXORD

```
DEST[31:0] := SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[MAX_KL-1:32] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

KXORW __mmask16 _mm512_kxor(__mmask16 a, __mmask16 b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-65, "TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)."

LAHF—Load Status Flags Into AH Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
9F	LAHF	ZO	Valid[1]	Valid	Load: AH := EFLAGS(SF:ZF:0:AF:0:PF:1:CF).

NOTES:

1. Invalid if CPUID.80000001H:ECX.LAHF_SAHF_64 is enumerated as 0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

This instruction executes as described above in compatibility mode and legacy mode. It is valid in 64-bit mode only if CPUID.80000001H:ECX.LAHF_SAHF_64[0] = 1.

Operation

```
IF 64-Bit Mode
    THEN
        IF CPUID.80000001H:ECX.LAHF_SAHF_64[0] = 1;
            THEN AH := RFLAGS(SF:ZF:0:AF:0:PF:1:CF);
            ELSE #UD;
        FI;
    ELSE
        AH := EFLAGS(SF:ZF:0:AF:0:PF:1:CF);
FI;
```

Flags Affected

None. The state of the flags in the EFLAGS register is not affected.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If CPUID.80000001H:ECX.LAHF_SAHF_64[0] = 0.
	If the LOCK prefix is used.

LAR—Load Access Rights

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 02 /r	LAR r16, r16/m16	RM	Valid	Valid	Load access rights from specified descriptor.
0F 02 /r	LAR r32, r32/m16[1]	RM	Valid	Valid	Load access rights from specified descriptor.
REX.W + 0F 02/r	LAR r32, r64/m16[1]	RM	Valid	N.E.	Load access rights from specified descriptor.

NOTES:

1. Regardless of operand size, only bits 15:0 of a register source operand are used. Other bits are ignored.

Instruction Operand Encoding

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

Description

Loads the access rights from the segment descriptor specified by the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the EFLAGS register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. If the source operand is a memory address, only 16 bits of data are accessed. The destination operand is a general-purpose register.

The processor performs access checks as part of the loading process. Once loaded in the destination register, software can perform additional checks on the access rights information.

The access rights for a segment descriptor include fields located in the second doubleword (bytes 4–7) of the segment descriptor. The following fields are loaded by the LAR instruction:

- Bits 7:0 are returned as 0
- Bits 11:8 return the segment type.
- Bit 12 returns the S flag.
- Bits 14:13 return the DPL.
- Bit 15 returns the P flag.
- The following fields are returned only if the operand size is greater than 16 bits:
  - Bits 19:16 are undefined.
  - Bit 20 returns the software-available bit in the descriptor.
  - Bit 21 returns the L flag.
  - Bit 22 returns the D/B flag.
  - Bit 23 returns the G flag.
  - Bits 31:24 are returned as 0.

When the operand size is 16 bits, only the low 16 bits identified above are returned; the upper bits of the destination are unmodified. When the operand size is 32 bits, the 32-bit value identified above is loaded into the destination operand; the upper bits of the destination are cleared. When the operand is 64 bits, the 32-bit value is zero-extended to 64 bits and loaded into the destination operand. (The behavior with 32-bit and 64-bit operand sizes is identical.)

This instruction performs the following checks before it loads the access rights in the destination register:

- Checks that the segment selector is not NULL.
- Checks that the segment selector points to a descriptor that is within the limits of the GDT or LDT being accessed

- Checks that the descriptor type is valid for this instruction. All code and data segment descriptors are valid for (can be accessed with) the LAR instruction. The valid system segment and gate descriptor types are given in Table 3-55.
- If the segment is not a conforming code segment, it checks that the specified segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL of the segment selector).

If the segment descriptor cannot be accessed or is an invalid type for the instruction, the ZF flag is cleared and no access rights are loaded in the destination operand.

The LAR instruction can only be executed in protected mode and IA-32e mode.

Table 3-55. Segment and Gate Types

Type	Protected Mode		IA-32e Mode	
	Name	Valid	Name	Valid
0	Reserved	No	Reserved	No
1	Available 16-bit TSS	Yes	Reserved	No
2	LDT	Yes	LDT	Yes
3	Busy 16-bit TSS	Yes	Reserved	No
4	16-bit call gate	Yes	Reserved	No
5	16-bit/32-bit task gate	Yes	Reserved	No
6	16-bit interrupt gate	No	Reserved	No
7	16-bit trap gate	No	Reserved	No
8	Reserved	No	Reserved	No
9	Available 32-bit TSS	Yes	Available 64-bit TSS	Yes
A	Reserved	No	Reserved	No
B	Busy 32-bit TSS	Yes	Busy 64-bit TSS	Yes
C	32-bit call gate	Yes	64-bit call gate	Yes
D	Reserved	No	Reserved	No
E	32-bit interrupt gate	No	64-bit interrupt gate	No
F	32-bit trap gate	No	64-bit trap gate	No

Operation

```
IF Offset(SRC) > descriptor table limit
    THEN
        ZF := 0;
    ELSE
        SegmentDescriptor := descriptor referenced by SRC;
        IF SegmentDescriptor(Type) ≠ conforming code segment
        and (CPL > DPL) or (RPL > DPL)
        or SegmentDescriptor(Type) is not valid for instruction
            THEN
                ZF := 0;
            ELSE
                DEST := access rights from SegmentDescriptor as given in Description section;
                ZF := 1;
        FI;
FI;
```

Flags Affected

The ZF flag is set to 1 if the access rights are loaded successfully; otherwise, it is cleared to 0.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#UD	The LAR instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The LAR instruction cannot be executed in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

LDDQU—Load Unaligned Integer 128 Bits

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F F0 /r LDDQU xmm1, mem	RM	V/V	SSE3	Load unaligned data from mem and return double quadword in xmm1.
VEX.128.F2.0F.WIG F0 /r VLDDQU xmm1, m128	RM	V/V	AVX	Load unaligned packed integer values from mem to xmm1.
VEX.256.F2.0F.WIG F0 /r VLDDQU ymm1, m256	RM	V/V	AVX	Load unaligned packed integer values from mem to ymm1.

Instruction Operand Encoding

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

Description

The instruction is *functionally similar* to (V)MOVDQU ymm/xmm, m256/m128 for loading from memory. That is: 32/16 bytes of data starting at an address specified by the source memory operand (second operand) are fetched from memory and placed in a destination register (first operand). The source operand need not be aligned on a 32/16-byte boundary. Up to 64/32 bytes may be loaded from memory; this is implementation dependent.

This instruction may improve performance relative to (V)MOVDQU if the source operand crosses a cache line boundary. In situations that require the data loaded by (V)LDDQU be modified and stored to the same location, use (V)MOVDQU or (V)MOVDQA instead of (V)LDDQU. To move a double quadword to or from memory locations that are known to be aligned on 16-byte boundaries, use the (V)MOVDQA instruction.

Implementation Notes

- If the source is aligned to a 32/16-byte boundary, based on the implementation, the 32/16 bytes may be loaded more than once. For that reason, the usage of (V)LDDQU should be avoided when using uncached or write-combining (WC) memory regions. For uncached or WC memory regions, keep using (V)MOVDQU.
- This instruction is a replacement for (V)MOVDQU (load) in situations where cache line splits significantly affect performance. It should not be used in situations where store-load forwarding is performance critical. If performance of store-load forwarding is critical to the application, use (V)MOVDQA store-load pairs when data is 256/128-bit aligned or (V)MOVDQU store-load pairs when data is 256/128-bit unaligned.
- If the memory address is not aligned on 32/16-byte boundary, some implementations may load up to 64/32 bytes and return 32/16 bytes in the destination. Some processor implementations may issue multiple loads to access the appropriate 32/16 bytes. Developers of multi-threaded or multi-processor software should be aware that on these processors the loads will be performed in a non-atomic way.
- If alignment checking is enabled (CR0.AM = 1, RFLAGS.AC = 1, and CPL = 3), an alignment-check exception (#AC) may or may not be generated (depending on processor implementation) when the memory address is not aligned on an 8-byte boundary.

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

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

Operation

LDDQU (128-bit Legacy SSE Version)

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

VLDDQU (VEX.128 Encoded Version)
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] := 0

VLDDQU (VEX.256 Encoded Version)
DEST[255:0] := SRC[255:0]
DEST[MAXVL-1:256] := 0

Intel C/C++ Compiler Intrinsic Equivalent

LDDQU __m128i _mm_lddqu_si128 (__m128i * p);
VLDDQU __m256i _mm256_lddqu_si256 (__m256i * p);

Numeric Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions."
Note treatment of #AC varies.

LDMXCSR—Load MXCSR Register

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
NP 0F AE /2 LDMXCSR m32	M	V/V	SSE	Load MXCSR register from m32.
VEX.LZ.0F.WIG AE /2 VLDMXCSR m32	M	V/V	AVX	Load MXCSR register from m32.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

Description

Loads the source operand into the MXCSR control/status register. The source operand is a 32-bit memory location. See "MXCSR Control and Status Register" in Chapter 10, of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for a description of the MXCSR register and its contents.

The LDMXCSR instruction is typically used in conjunction with the (V)STMXCSR instruction, which stores the contents of the MXCSR register in memory.

The default MXCSR value at reset is 1F80H.

If a (V)LDMXCSR instruction clears a SIMD floating-point exception mask bit and sets the corresponding exception flag bit, a SIMD floating-point exception will not be immediately generated. The exception will be generated only upon the execution of the next instruction that meets both conditions below:

- the instruction must operate on an XMM or YMM register operand,
- the instruction causes that particular SIMD floating-point exception to be reported.

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

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

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

Operation

```
MXCSR := m32;
```

C/C++ Compiler Intrinsic Equivalent

_mm_setcsr(unsigned int i)

Numeric Exceptions

None.

Other Exceptions

See Table 2-22, "Type 5 Class Exception Conditions," additionally:

#GP	For an attempt to set reserved bits in MXCSR.
#UD	If VEX.vvvv ≠ 1111B.

LDS/LES/LFS/LGS/LSS—Load Far Pointer

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
C5 /r	LDS r16,m16:16	RM	Invalid	Valid	Load DS:r16 with far pointer from memory.
C5 /r	LDS r32,m16:32	RM	Invalid	Valid	Load DS:r32 with far pointer from memory.
0F B2 /r	LSS r16,m16:16	RM	Valid	Valid	Load SS:r16 with far pointer from memory.
0F B2 /r	LSS r32,m16:32	RM	Valid	Valid	Load SS:r32 with far pointer from memory.
REX.W + 0F B2 /r	LSS r64,m16:64	RM	Valid	N.E.	Load SS:r64 with far pointer from memory.
C4 /r	LES r16,m16:16	RM	Invalid	Valid	Load ES:r16 with far pointer from memory.
C4 /r	LES r32,m16:32	RM	Invalid	Valid	Load ES:r32 with far pointer from memory.
0F B4 /r	LFS r16,m16:16	RM	Valid	Valid	Load FS:r16 with far pointer from memory.
0F B4 /r	LFS r32,m16:32	RM	Valid	Valid	Load FS:r32 with far pointer from memory.
REX.W + 0F B4 /r	LFS r64,m16:64	RM	Valid	N.E.	Load FS:r64 with far pointer from memory.
0F B5 /r	LGS r16,m16:16	RM	Valid	Valid	Load GS:r16 with far pointer from memory.
0F B5 /r	LGS r32,m16:32	RM	Valid	Valid	Load GS:r32 with far pointer from memory.
REX.W + 0F B5 /r	LGS r64,m16:64	RM	Valid	N.E.	Load GS:r64 with far pointer from memory.

Instruction Operand Encoding

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

Description

Loads a far pointer (segment selector and offset) from the second operand (source operand) into a segment register and the first operand (destination operand). The source operand specifies a 48-bit or a 32-bit pointer in memory depending on the current setting of the operand-size attribute (32 bits or 16 bits, respectively). The instruction opcode and the destination operand specify a segment register/general-purpose register pair. The 16-bit segment selector from the source operand is loaded into the segment register specified with the opcode (DS, SS, ES, FS, or GS). The 32-bit or 16-bit offset is loaded into the register specified with the destination operand.

If one of these instructions is executed in protected mode, additional information from the segment descriptor pointed to by the segment selector in the source operand is loaded in the hidden part of the selected segment register.

Also in protected mode, a NULL selector (values 0000 through 0003) can be loaded into DS, ES, FS, or GS registers without causing a protection exception. (Any subsequent reference to a segment whose corresponding segment register is loaded with a NULL selector, causes a general-protection exception (#GP) and no memory reference to the segment occurs.)

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.W promotes operation to specify a source operand referencing an 80-bit pointer (16-bit selector, 64-bit offset) in memory. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
64-BIT_MODE
    IF SS is loaded
        THEN
            IF SegmentSelector = NULL and ( (RPL = 3) or
                    (RPL ≠ 3 and RPL ≠ CPL) )
                THEN #GP(0);
            ELSE IF descriptor is in non-canonical space
```

```
            THEN #GP(selector); FI;
        ELSE IF Segment selector index is not within descriptor table limits
                or segment selector RPL ≠ CPL
                or access rights indicate nonwritable data segment
                or DPL ≠ CPL
            THEN #GP(selector); FI;
        ELSE IF Segment marked not present
            THEN #SS(selector); FI;
        FI;
        SS := SegmentSelector(SRC);
        SS := SegmentDescriptor([SRC]);
ELSE IF attempt to load DS, or ES
    THEN #UD;
ELSE IF FS, or GS is loaded with non-NULL segment selector
    THEN IF Segment selector index is not within descriptor table limits
        or access rights indicate segment neither data nor readable code segment
        or segment is data or nonconforming-code segment
        and ( RPL > DPL or CPL > DPL)
            THEN #GP(selector); FI;
        ELSE IF Segment marked not present
            THEN #NP(selector); FI;
        FI;
        SegmentRegister := SegmentSelector(SRC) ;
        SegmentRegister := SegmentDescriptor([SRC]);
    FI;
ELSE IF FS, or GS is loaded with a NULL selector:
    THEN
        SegmentRegister := NULLSelector;
        SegmentRegister(DescriptorValidBit) := 0; FI; (* Hidden flag;
            not accessible by software *)
FI;
DEST := Offset(SRC);
PREOTECTED MODE OR COMPATIBILITY MODE;
IF SS is loaded
    THEN
        IF SegementSelector = NULL
            THEN #GP(0);
        ELSE IF Segment selector index is not within descriptor table limits
                or segment selector RPL ≠ CPL
                or access rights indicate nonwritable data segment
                or DPL ≠ CPL
            THEN #GP(selector); FI;
        ELSE IF Segment marked not present
            THEN #SS(selector); FI;
        FI;
        SS := SegmentSelector(SRC);
        SS := SegmentDescriptor([SRC]);
ELSE IF DS, ES, FS, or GS is loaded with non-NULL segment selector
    THEN IF Segment selector index is not within descriptor table limits
        or access rights indicate segment neither data nor readable code segment
        or segment is data or nonconforming-code segment
        and (RPL > DPL or CPL > DPL)
            THEN #GP(selector); FI;
        ELSE IF Segment marked not present
```

```
                THEN #NP(selector); FI;
            FI;
            SegmentRegister := SegmentSelector(SRC) AND RPL;
            SegmentRegister := SegmentDescriptor([SRC]);
        FI;
    ELSE IF DS, ES, FS, or GS is loaded with a NULL selector:
        THEN
            SegmentRegister := NULLSelector;
            SegmentRegister(DescriptorValidBit) := 0; FI; (* Hidden flag;
                not accessible by software *)
    FI;
    DEST := Offset(SRC);
Real-Address or Virtual-8086 Mode
    SegmentRegister := SegmentSelector(SRC); FI;
    DEST := Offset(SRC);
```

Flags Affected

None.

Protected Mode Exceptions

#UD	If source operand is not a memory location.
	If the LOCK prefix is used.
#GP(0)	If a NULL selector is loaded into the SS register.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#GP(selector)	If the SS register is being loaded and any of the following is true: the segment selector index is not within the descriptor table limits, the segment selector RPL is not equal to CPL, the segment is a non-writable data segment, or DPL is not equal to CPL.
	If the DS, ES, FS, or GS register is being loaded with a non-NULL segment selector and any of the following is true: the segment selector index is not within descriptor table limits, the segment is neither a data nor a readable code segment, or the segment is a data or noncon-forming-code segment and both RPL and CPL are greater than DPL.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If the SS register is being loaded and the segment is marked not present.
#NP(selector)	If DS, ES, FS, or GS register is being loaded with a non-NULL segment selector and the segment is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If source operand is not a memory location.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD	If source operand is not a memory location.
	If the LOCK prefix is used.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
	If a NULL selector is attempted to be loaded into the SS register in compatibility mode.
	If a NULL selector is attempted to be loaded into the SS register in CPL3 and 64-bit mode.
	If a NULL selector is attempted to be loaded into the SS register in non-CPL3 and 64-bit mode where its RPL is not equal to CPL.
#GP(Selector)	If the FS, or GS register is being loaded with a non-NULL segment selector and any of the following is true: the segment selector index is not within descriptor table limits, the memory address of the descriptor is non-canonical, the segment is neither a data nor a readable code segment, or the segment is a data or nonconforming-code segment and both RPL and CPL are greater than DPL.
	If the SS register is being loaded and any of the following is true: the segment selector index is not within the descriptor table limits, the memory address of the descriptor is non-canonical, the segment selector RPL is not equal to CPL, the segment is a nonwritable data segment, or DPL is not equal to CPL.
#SS(0)	If a memory operand effective address is non-canonical
#SS(Selector)	If the SS register is being loaded and the segment is marked not present.
#NP(selector)	If FS, or GS register is being loaded with a non-NULL segment selector and the segment is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If source operand is not a memory location.
	If the LOCK prefix is used.

LDTILECFG—Load Tile Configuration

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.NP.0F38.W0 49 !(11):000:bbb LDTILECFG m512	A	V/N.E.	AMX_TILE	Load tile configuration as specified in m512.

Instruction Operand Encoding

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

Description

The LDTILECFG instruction takes an operand containing a pointer to a 64-byte memory location containing the description of the tiles to be supported. In order to configure the tiles, the AMX_TILE bit in CPUID must be set and the operating system has to have enabled the tiles architecture.

The memory area contains the palette and describes how many tiles are being used and defines each tile in terms of rows and column bytes. Requests must be compatible with the restrictions provided by CPUID; see Table 3-56 below.

Table 3-56. Memory Area Layout

Byte(s)	Field Name	Description
0	palette	Palette selects the supported configuration of the tiles that will be used.
1	start_row	start_row is used for storing the restart values for interrupted operations.
2-15	reserved, must be zero	
16-17	tile0.colsb	Tile 0 bytes per row.
18-19	tile1.colsb	Tile 1 bytes per row.
20-21	tile2.colsb	Tile 2 bytes per row.
...	(sequence continues)	
30-31	tile7.colsb	Tile 7 bytes per row.
32-47	reserved, must be zero	
48	tile0.rows	Tile 0 rows.
49	tile1.rows	Tile 1 rows.
50	tile2.rows	Tile 2 rows.
...	(sequence continues)	
55	tile7.rows	Tile 7 rows.
56-63	reserved, must be zero	

If a tile row and column pair is not used to specify tile parameters, they must have the value zero. All enabled tiles (based on the palette) must be configured. Specifying tile parameters for more tiles than the implementation limit or the palette limit results in a #GP fault.

If the palette_id is zero, that signifies the INIT state for both TILECFG and TILEDATA. Tiles are zeroed in the INIT state. The only legal non-INIT value for palette_id is 1.

Any attempt to execute the LDTILECFG instruction inside an Intel TSX transaction will result in a transaction abort.

Operation

LDTILECFG mem

```
error := False
buf := read_memory(mem, 64)
temp_tilecfg.palette_id := buf.byte[0]
if temp_tilecfg.palette_id > max_palette:
    error := True
if not xcr0_supports_palette(temp_tilecfg.palette_id):
    error := True
if temp_tilecfg.palette_id !=0:
    temp_tilecfg.start_row := buf.byte[1]
    if buf.byte[2..15] is nonzero:
        error := True
    p := 16
    # configure columns
    for n in 0 ... palette_table[temp_tilecfg.palette_id].max_names-1:
        temp_tilecfg.t[n].colsb:= buf.word[p/2]
        p := p + 2
        if temp_tilecfg.t[n].colsb > palette_table[temp_tilecfg.palette_id].bytes_per_row:
            error := True
    if nonzero(buf[p...47]):
        error := True

    # configure rows
    p := 48
    for n in 0 ... palette_table[temp_tilecfg.palette_id].max_names-1:
        temp_tilecfg.t[n].rows:= buf.byte[p]
        if temp_tilecfg.t[n].rows > palette_table[temp_tilecfg.palette_id].max_rows:
            error := True
        p := p + 1

    if nonzero(buf[p...63]):
        error := True

    # validate each tile's row & col configs are reasonable and enable the valid tiles
    for n in 0 ... palette_table[temp_tilecfg.palette_id].max_names-1:
        if temp_tilecfg.t[n].rows !=0 and temp_tilecfg.t[n].colsb != 0:
            temp_tilecfg.t[n].valid := 1
        elif temp_tilecfg.t[n].rows == 0 and temp_tilecfg.t[n].colsb == 0:
            temp_tilecfg.t[n].valid := 0
        else:
            error := True// one of rows or colsbwas 0 but not both.

if error:
    #GP
elif temp_tilecfg.palette_id == 0:
    TILES_CONFIGURED := 0// init state
    tilecfg := 0// equivalent to 64B of zeros
    zero_all_tile_data()
else:
    tilecfg := temp_tilecfg
    zero_all_tile_data()
    TILES_CONFIGURED := 1
```

Intel C/C++ Compiler Intrinsic Equivalent

LDTILECFG void _tile_loadconfig(const void *);

Flags Affected

None.

Exceptions

AMX-E1; see Section 2.10, "Intel® AMX Instruction Exception Classes," for details.

LEA—Load Effective Address

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
8D /r	LEA r16,m	RM	Valid	Valid	Store effective address for m in register r16.
8D /r	LEA r32,m	RM	Valid	Valid	Store effective address for m in register r32.
REX.W + 8D /r	LEA r64,m	RM	Valid	N.E.	Store effective address for m in register r64.

Instruction Operand Encoding

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

Description

Computes the effective address of the second operand (the source operand) and stores it in the first operand (destination operand). The source operand is a memory address (offset part) specified with one of the processors addressing modes; the destination operand is a general-purpose register. The address-size and operand-size attributes affect the action performed by this instruction, as shown in the following table. The operand-size attribute of the instruction is determined by the chosen register; the address-size attribute is determined by the attribute of the code segment.

Table 3-57. Non-64-bit Mode LEA Operation with Address and Operand Size Attributes

Operand Size	Address Size	Action Performed
16	16	16-bit effective address is calculated and stored in requested 16-bit register destination.
16	32	32-bit effective address is calculated. The lower 16 bits of the address are stored in the requested 16-bit register destination.
32	16	16-bit effective address is calculated. The 16-bit address is zero-extended and stored in the requested 32-bit register destination.
32	32	32-bit effective address is calculated and stored in the requested 32-bit register destination.

Different assemblers may use different algorithms based on the size attribute and symbolic reference of the source operand.

In 64-bit mode, the instruction's destination operand is governed by operand size attribute, the default operand size is 32 bits. Address calculation is governed by address size attribute, the default address size is 64-bits. In 64-bit mode, address size of 16 bits is not encodable. See Table 3-58.

Table 3-58. 64-bit Mode LEA Operation with Address and Operand Size Attributes

Operand Size	Address Size	Action Performed
16	32	32-bit effective address is calculated (using 67H prefix). The lower 16 bits of the address are stored in the requested 16-bit register destination (using 66H prefix).
16	64	64-bit effective address is calculated (default address size). The lower 16 bits of the address are stored in the requested 16-bit register destination (using 66H prefix).
32	32	32-bit effective address is calculated (using 67H prefix) and stored in the requested 32-bit register destination.
32	64	64-bit effective address is calculated (default address size) and the lower 32 bits of the address are stored in the requested 32-bit register destination.
64	32	32-bit effective address is calculated (using 67H prefix), zero-extended to 64-bits, and stored in the requested 64-bit register destination (using REX.W).
64	64	64-bit effective address is calculated (default address size) and all 64-bits of the address are stored in the requested 64-bit register destination (using REX.W).

Operation

```
IF OperandSize = 16 and AddressSize = 16
    THEN
        DEST := EffectiveAddress(SRC); (* 16-bit address *)
    ELSE IF OperandSize = 16 and AddressSize = 32
        THEN
            temp := EffectiveAddress(SRC); (* 32-bit address *)
            DEST := temp[0:15]; (* 16-bit address *)
        FI;
    ELSE IF OperandSize = 32 and AddressSize = 16
        THEN
            temp := EffectiveAddress(SRC); (* 16-bit address *)
            DEST := ZeroExtend(temp); (* 32-bit address *)
        FI;
    ELSE IF OperandSize = 32 and AddressSize = 32
        THEN
            DEST := EffectiveAddress(SRC); (* 32-bit address *)
        FI;
    ELSE IF OperandSize = 16 and AddressSize = 64
        THEN
            temp := EffectiveAddress(SRC); (* 64-bit address *)
            DEST := temp[0:15]; (* 16-bit address *)
        FI;
    ELSE IF OperandSize = 32 and AddressSize = 64
        THEN
            temp := EffectiveAddress(SRC); (* 64-bit address *)
            DEST := temp[0:31]; (* 16-bit address *)
        FI;
    ELSE IF OperandSize = 64 and AddressSize = 64
        THEN
            DEST := EffectiveAddress(SRC); (* 64-bit address *)
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#UD	If source operand is not a memory location. If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

LEAVE—High Level Procedure Exit

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
C9	LEAVE	ZO	Valid	Valid	Set SP to BP, then pop BP.
C9	LEAVE	ZO	N.E.	Valid	Set ESP to EBP, then pop EBP.
C9	LEAVE	ZO	Valid	N.E.	Set RSP to RBP, then pop RBP.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Releases the stack frame set up by an earlier ENTER instruction. The LEAVE instruction copies the frame pointer (in the EBP register) into the stack pointer register (ESP), which releases the stack space allocated to the stack frame. The old frame pointer (the frame pointer for the calling procedure that was saved by the ENTER instruction) is then popped from the stack into the EBP register, restoring the calling procedure's stack frame.

A RET instruction is commonly executed following a LEAVE instruction to return program control to the calling procedure.

See "Procedure Calls for Block-Structured Languages" in Chapter 6 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for detailed information on the use of the ENTER and LEAVE instructions.

In 64-bit mode, the instruction's default operation size is 64 bits; 32-bit operation cannot be encoded. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF StackAddressSize = 32
    THEN
        ESP := EBP;
    ELSE IF StackAddressSize = 64
        THEN RSP := RBP; FI;
    ELSE IF StackAddressSize = 16
        THEN SP := BP; FI;
FI;

IF OperandSize = 32
    THEN EBP := Pop();
    ELSE IF OperandSize = 64
        THEN RBP := Pop(); FI;
    ELSE IF OperandSize = 16
        THEN BP := Pop(); FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#SS(0)	If the EBP register points to a location that is not within the limits of the current stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If the EBP register points to a location outside of the effective address space from 0 to FFFFH.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If the EBP register points to a location outside of the effective address space from 0 to FFFFH.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If the stack address is in a non-canonical form.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

LFENCE—Load Fence

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F AE E8 LFENCE	ZO	V/V	SSE2	Serializes load operations.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Performs a serializing operation on all load-from-memory instructions that were issued prior the LFENCE instruction. Specifically, LFENCE does not execute until all prior instructions have completed locally, and no later instruction begins execution until LFENCE completes. In particular, an instruction that loads from memory and that precedes an LFENCE receives data from memory prior to completion of the LFENCE. (An LFENCE that follows an instruction that stores to memory might complete **before** the data being stored have become globally visible.) Instructions following an LFENCE may be fetched from memory before the LFENCE, but they will not execute (even speculatively) until the LFENCE completes.

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

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

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

Specification of the instruction's opcode above indicates a ModR/M byte of E8. For this instruction, the processor ignores the r/m field of the ModR/M byte. Thus, LFENCE is encoded by any opcode of the form 0F AE Ex, where x is in the range 8-F.

Operation

```
Wait_On_Following_Instructions_Until(preceding_instructions_complete);
```

Intel C/C++ Compiler Intrinsic Equivalent

void _mm_lfence(void)

Exceptions (All Modes of Operation)

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

LGDT/LIDT—Load Global/Interrupt Descriptor Table Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 /2	LGDT m16&32	M	N.E.	Valid	Load m into GDTR.
0F 01 /3	LIDT m16&32	M	N.E.	Valid	Load m into IDTR.
0F 01 /2	LGDT m16&64	M	Valid	N.E.	Load m into GDTR.
0F 01 /3	LIDT m16&64	M	Valid	N.E.	Load m into IDTR.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

Description

Loads the values in the source operand into the global descriptor table register (GDTR) or the interrupt descriptor table register (IDTR). The source operand specifies a 6-byte memory location that contains the base address (a linear address) and the limit (size of table in bytes) of the global descriptor table (GDT) or the interrupt descriptor table (IDT). If operand-size attribute is 32 bits, a 16-bit limit (lower 2 bytes of the 6-byte data operand) and a 32-bit base address (upper 4 bytes of the data operand) are loaded into the register. If the operand-size attribute is 16 bits, a 16-bit limit (lower 2 bytes) and a 24-bit base address (third, fourth, and fifth byte) are loaded. Here, the high-order byte of the operand is not used and the high-order byte of the base address in the GDTR or IDTR is filled with zeros.

The LGDT and LIDT instructions are used only in operating-system software; they are not used in application programs. They are the only instructions that directly load a linear address (that is, not a segment-relative address) and a limit in protected mode. They are commonly executed in real-address mode to allow processor initialization prior to switching to protected mode.

In 64-bit mode, the instruction's operand size is fixed at 8+2 bytes (an 8-byte base and a 2-byte limit). See the summary chart at the beginning of this section for encoding data and limits.

See "SGDT—Store Global Descriptor Table Register" in Chapter 4, of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B, for information on storing the contents of the GDTR and IDTR.

Operation

```
IF Instruction is LIDT
    THEN
        IF OperandSize = 16
            THEN
                IDTR(Limit) := SRC[0:15];
                IDTR(Base) := SRC[16:47] AND 00FFFFFFH;
            ELSE IF 32-bit Operand Size
                THEN
                    IDTR(Limit) := SRC[0:15];
                    IDTR(Base) := SRC[16:47];
                FI;
            ELSE IF 64-bit Operand Size (* In 64-Bit Mode *)
                THEN
                    IDTR(Limit) := SRC[0:15];
                    IDTR(Base) := SRC[16:79];
                FI;
        FI;
    ELSE (* Instruction is LGDT *)
        IF OperandSize = 16
            THEN
                GDTR(Limit) := SRC[0:15];
                GDTR(Base) := SRC[16:47] AND 00FFFFFFH;
            ELSE IF 32-bit Operand Size
                THEN
                    GDTR(Limit) := SRC[0:15];
                    GDTR(Base) := SRC[16:47];
                FI;
            ELSE IF 64-bit Operand Size (* In 64-Bit Mode *)
                THEN
                    GDTR(Limit) := SRC[0:15];
                    GDTR(Base) := SRC[16:79];
                FI;
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
#GP(0)	If the current privilege level is not 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.
#GP	If the current privilege level is not 0.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the current privilege level is not 0.
	If the memory address is in a non-canonical form.
#UD	If the LOCK prefix is used.
#PF(fault-code)	If a page fault occurs.

LKGS—Load Kernel GS Base

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F2 0F 00 /6	LKGS r/m16	A	Valid	N.E.	Load GS using r/m16, placing the base address instead in the IA32_KERNEL_GS_BASE MSR.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	16-bit selector	N/A	N/A	N/A

Description

LKGS operates in the same way as MOV to GS except that the descriptor's base address is loaded into the IA32_KERNEL_GS_BASE MSR instead of the GS segment's descriptor cache.

LKGS takes a single (source) operand, which can be a general-purpose register or a memory location. The operand must be a valid segment selector. The instruction loads the segment descriptor referenced by that segment selector into the GS descriptor cache, with the exception of the base address. The base address in the GS descriptor cache is not modified; the base address from the segment descriptor is loaded into the IA32_KERNEL_GS_BASE MSR. (Since the base address in the descriptor is only 32 bits, the upper 32 bits of the MSR are cleared.)

A null segment selector (values 0000-0003) can be loaded without causing an exception. However, any subsequent attempt to reference GS outside 64-bit mode causes a general protection exception (#GP) and no memory reference occurs. LKGS with a null segment selector loads zero into IA32_KERNEL_GS_BASE.

Operation

```
IF CPL > 0 OR logical processor not in 64-bit mode
    THEN #UD; FI;
IF SRC is null
    THEN
        GS.selector := SRC;
        mark GS as null;
        IA32_KERNEL_GS_BASE := 0;
    ELSE
        IF SRC.index is outside descriptor table limits
            THEN #GP(selector); FI;
        read referenced descriptor for descriptor table;
        IF the descriptor is not for a data or readable code segment OR SRC.RPL > descriptor.DPL
            THEN #GP(selector); FI;
        IF the descriptor is not marked present
            THEN #NP(selector);
            ELSE
                GS.selector := SRC;
                GS.attributes := descriptor.attributes;
                IA32_KERNEL_GS_BASE := descriptor.base; // bits 63:32 cleared
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#UD	The LKGS instruction is not recognized in protected mode.

Real-Address Mode Exceptions

#UD	The LKGS instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The LKGS instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	The LKGS instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD	If CPL > 0.
	If CPUID.07H.01H:EAX.LKGS[18] is not 1.
	If the LOCK prefix is used.
#GP(0)	If the memory address is in a non-canonical form.
#GP(selector)	If segment selector index is outside descriptor table limits.
	If the memory access to the descriptor table is non-canonical.
	If the referenced descriptor is not for a data or readable code segment or the selector RPL is greater than the descriptor DPL.
#SS(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.

LLDT—Load Local Descriptor Table Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 00 /2	LLDT r/m16	M	Valid	Valid	Load LDTR using segment selector in r/m16.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

Description

Loads the source operand into the segment selector field of the local descriptor table register (LDTR). The source operand (a general-purpose register or a memory location) contains a segment selector that references an LDT descriptor in the global descriptor table (GDT) that in turn references a local descriptor table (LDT). The instruction loads LDTR with the segment limit and base address for the LDT from the LDT descriptor. The segment registers DS, ES, SS, FS, GS, and CS are not affected by this instruction, nor is the LDTR field in the task state segment (TSS) for the current task.

If bits 15:2 of the source operand are 0, LDTR is marked invalid and the LLDT instruction completes without fault. However, any subsequent reference to a descriptor in the LDT (except by the LAR, VERR, VERW or LSL instructions) causes a general protection exception (#GP).

The operand-size attribute has no effect on this instruction.

The LLDT instruction is provided for use in operating-system software; it cannot be used in application programs. It can be executed only when the CPL is 0. This instruction cannot be executed in real-address mode or in virtual-8086 mode.

In 64-bit mode, the operand size is still fixed at 16 bits. The instruction uses a 16-byte LDT descriptor that contains a 64-bit base address.

Operation

```
IF SRC is a NULL selector
    THEN LDTR := INVALID;
ELSE IF SRC(type) ≠ global OR SRC(Offset) > GDTR.limit
    THEN #GP(SRC);
ELSE
    Read SegmentDescriptor from GDT;
    IF SegmentDescriptor(Type) ≠ LDT
        THEN #GP(segment selector); FI;
    IF segment descriptor is not present
        THEN #NP(segment selector); FI;
    LDTR(SegmentSelector) := SRC;
    LDTR(SegmentDescriptor) := GDTSegmentDescriptor;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and contains a NULL segment selector.
#GP(selector)	If the selector operand does not point into the GDT or if the entry in the GDT is not an LDT descriptor.
	If the selector points to the LDT or is beyond the GDT limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NP(selector)	If the LDT descriptor is not present.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	The LLDT instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The LLDT instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the current privilege level is not 0.
	If the memory address is in a non-canonical form.
#GP(selector)	If the source selector does not point into the GDT or if the entry in the GDT is not an LDT descriptor.
	If the source segment selector is beyond the GDT limit.
#NP(selector)	If the LDT descriptor is not present.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

LMSW—Load Machine Status Word

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 /6	LMSW r/m16	M	Valid	Valid	Loads r/m16 in machine status word of CR0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

Description

Loads the source operand into the machine status word, bits 0 through 15 of register CR0. The source operand can be a 16-bit general-purpose register or a memory location. Only the low-order 4 bits of the source operand (which contains the PE, MP, EM, and TS flags) are loaded into CR0. The PG, CD, NW, AM, WP, NE, and ET flags of CR0 are not affected. The operand-size attribute has no effect on this instruction.

If the PE flag of the source operand (bit 0) is set to 1, the instruction causes the processor to switch to protected mode. While in protected mode, the LMSW instruction cannot be used to clear the PE flag and force a switch back to real-address mode.

The LMSW instruction is provided for use in operating-system software; it should not be used in application programs. In protected or virtual-8086 mode, it can only be executed at CPL 0.

This instruction is provided for compatibility with the Intel 286 processor; programs and procedures intended to run on IA-32 and Intel 64 processors beginning with Intel386 processors should use the MOV (control registers) instruction to load the whole CR0 register. The MOV CR0 instruction can be used to set and clear the PE flag in CR0, allowing a procedure or program to switch between protected and real-address modes.

This instruction is a serializing instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode. Note that the operand size is fixed at 16 bits.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 27 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C, for more information about the behavior of this instruction in VMX non-root operation.

Operation

```
CR0[0:3] := SRC[0:3];
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

#GP(0)	The LMSW instruction is not recognized in virtual-8086 mode.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the current privilege level is not 0.
	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

LOADIWKEY—Load Internal Wrapping Key With Key Locker

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 38 DC 11:rrr:bbb LOADIWKEY xmm1, xmm2, <EAX>, <XMM0>	A	V/V	KEY_LOCKER	Load internal wrapping key from xmm1, xmm2, and XMM0.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r)	ModRM:r/m (r)	Implicit EAX (r)	Implicit XMM0 (r)

Description

The LOADIWKEY[1] instruction writes the Key Locker internal wrapping key, which is called IWKey. This IWKey is used by the ENCODEKEY* instructions to wrap keys into handles. Conversely, the AESENC/DEC*KL instructions use IWKey to unwrap those keys from the handles and help verify the handle integrity. For security reasons, no instruction is designed to allow software to directly read the IWKey value.

IWKey includes two cryptographic keys as well as metadata. The two cryptographic keys are loaded from register sources so that LOADIWKEY can be executed without the keys ever being in memory.

The key input operands are:

- The 256-bit encryption key is loaded from the two explicit operands.
- The 128-bit integrity key is loaded from the implicit operand XMM0.

The implicit operand EAX specifies the KeySource and whether backing up the key is permitted:

- EAX[0] – When set, the wrapping key being initialized is not permitted to be backed up to platform-scoped storage.
- EAX[4:1] – This specifies the KeySource, which is the type of key. Currently only two encodings are supported. A KeySource of 0 indicates that the key input operands described above should be directly stored as the internal wrapping keys. LOADIWKEY with a KeySource of 1 will have random numbers from the on-chip random number generator XORed with the source registers (including XMM0) so that the software that executes the LOADIWKEY does not know the actual IWKey encryption and integrity keys. Software can choose to put additional random data into the source registers so that other sources of random data are combined with the hardware random number generator supplied value. Software should always check ZF after executing LOADIWKEY with KeySource of 1 as this operation may fail due to it being unable to get sufficient full-entropy data from the on-chip random number generator. Both KeySource of 0 and 1 specify that IWKey be used with the AES-GCM-SIV algorithm. CPUID.19H:ECX[1] enumerates support for KeySource of 1. All other KeySource encodings are reserved.
- EAX[31:5] – Reserved.

1. Further details on Key Locker and usage of this instruction can be found here: https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html.

Operation

LOADIWKEY

```
IF CPL > 0                    // LOADKWKEY only allowed at ring 0 (supervisor mode)
    THEN #GP (0); FI;
IF EAX[4:1] > 1               // Reserved KeySource encoding used
    THEN #GP (0); FI;
IF EAX[31:5] != 0             // Reserved bit in EAX is set
    THEN #GP (0); FI;
IF EAX[0] AND (CPUID.19H:ECX[0] == 0) // NoBackup is not supported on this part
    THEN #GP (0); FI;
IF (EAX[4:1] == 1) AND (CPUID.19H:ECX[1] == 0) // KeySource of 1 is not supported on this part
    THEN #GP (0); FI;
IF (EAX[4:1] == 0)      // KeySource of 0
    THEN
        IWKey.Encryption Key[127:0] := SRC2[127:0]:
        IWKey.Encryption Key[255:128] := SRC1[127:0];
        IWKey.IntegrityKey[127:0] := XMM0[127:0];
        IWKey.NoBackup = EAX [0];
        IWKey.KeySource = EAX [4:1];
        RFLAGS.ZF := 0;
    ELSE                      // KeySource of 1. See RDSEED definition for details of randomness
        IF HW_NRND_GEN.ready == 1          // Full-entropy random data from RDSEED hardware block was received
            THEN
                IWKey.Encryption Key[127:0] := SRC2[127:0] XOR HW_NRND_GEN.data[127:0];
                IWKey.Encryption Key[255:128] := SRC1[127:0] XOR HW_NRND_GEN.data[255:128];
                IWKey.IntegrityKey[127:0] := XMM0[127:0] XOR HW_NRND_GEN.data[383:256];
                IWKey.NoBackup = EAX [0];
                IWKey.KeySource = EAX [4:1];
                RFLAGS.ZF := 0;
            ELSE          // Random data was not returned from RDSEED hardware block. IWKey was not loaded
                RFLAGS.ZF := 1;
        FI;
FI;
RFLAGS.OF, SF, AF, PF, CF := 0;
```

Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to full-entropy random data not being received from RDSEED. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

Intel C/C++ Compiler Intrinsic Equivalent

LOADIWKEY void _mm_loadiwkey(unsigned int ctl, __m128i intkey, __m128i enkey_lo, __m128i enkey_hi);

Exceptions (All Operating Modes)

#GP	If CPL > 0. (Does not apply in real-address mode.)
	If EAX[4:1] > 1.
	If EAX[31:5] != 0.
	If (EAX[0] == 1) AND (CPUID.19H:ECX[0] == 0).
	If (EAX[4:1] == 1) AND (CPUID.19H:ECX[1] == 0).
#UD	If the LOCK prefix is used.
	If CPUID.07H.00H:ECX.KEY_LOCKER[23] = 0.
	If CR4.KL = 0.
	If CR0.EM = 1.
	If CR4.OSFXSR = 0.
#NM	If CR0.TS = 1.

LOCK—Assert LOCK# Signal Prefix

Opcode[1]	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F0	LOCK	ZO	Valid	Valid	Asserts LOCK# signal for duration of the accompanying instruction.

NOTES:

1. See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Causes the processor's LOCK# signal to be asserted during execution of the accompanying instruction (turns the instruction into an atomic instruction). In a multiprocessor environment, the LOCK# signal ensures that the processor has exclusive use of any shared memory while the signal is asserted.

In most IA-32 and all Intel 64 processors, locking may occur without the LOCK# signal being asserted. See the "IA-32 Architecture Compatibility" section below for more details.

The LOCK prefix can be prepended only to the following instructions and only to those forms of the instructions where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, CMPXCHG16B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG. If the LOCK prefix is used with one of these instructions and the source operand is a memory operand, an undefined opcode exception (#UD) may be generated. An undefined opcode exception will also be generated if the LOCK prefix is used with any instruction not in the above list. The XCHG instruction always asserts the LOCK# signal regardless of the presence or absence of the LOCK prefix.

The LOCK prefix is typically used with the BTS instruction to perform a read-modify-write operation on a memory location in shared memory environment.

The integrity of the LOCK prefix is not affected by the alignment of the memory field. Memory locking is observed for arbitrarily misaligned fields.

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

IA-32 Architecture Compatibility

Beginning with the P6 family processors, when the LOCK prefix is prefixed to an instruction and the memory area being accessed is cached internally in the processor, the LOCK# signal is generally not asserted. Instead, only the processor's cache is locked. Here, the processor's cache coherency mechanism ensures that the operation is carried out atomically with regards to memory. See "Effects of a Locked Operation on Internal Processor Caches" in Chapter 11 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, the for more information on locking of caches.

Operation

```
AssertLOCK#(DurationOfAccompaningInstruction);
```

Flags Affected

None.

Protected Mode Exceptions

#UD	If the LOCK prefix is used with an instruction not listed: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, CMPXCHG16B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, XCHG. Other exceptions can be generated by the instruction when the LOCK prefix is applied.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

LODS/LODSB/LODSW/LODSD/LODSQ—Load String

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
AC	LODS m8	ZO	Valid	Valid	For legacy mode, Load byte at address DS:(E)SI into AL. For 64-bit mode load byte at address (R)SI into AL.
AD	LODS m16	ZO	Valid	Valid	For legacy mode, Load word at address DS:(E)SI into AX. For 64-bit mode load word at address (R)SI into AX.
AD	LODS m32	ZO	Valid	Valid	For legacy mode, Load dword at address DS:(E)SI into EAX. For 64-bit mode load dword at address (R)SI into EAX.
REX.W + AD	LODS m64	ZO	Valid	N.E.	Load qword at address (R)SI into RAX.
AC	LODSB	ZO	Valid	Valid	For legacy mode, Load byte at address DS:(E)SI into AL. For 64-bit mode load byte at address (R)SI into AL.
AD	LODSW	ZO	Valid	Valid	For legacy mode, Load word at address DS:(E)SI into AX. For 64-bit mode load word at address (R)SI into AX.
AD	LODSD	ZO	Valid	Valid	For legacy mode, Load dword at address DS:(E)SI into EAX. For 64-bit mode load dword at address (R)SI into EAX.
REX.W + AD	LODSQ	ZO	Valid	N.E.	Load qword at address (R)SI into RAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Loads a byte, word, or doubleword from the source operand into the AL, AX, or EAX register, respectively. The source operand is a memory location, the address of which is read from the DS:ESI or the DS:SI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The DS segment may be overridden with a segment override prefix.

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operands" form and the "no-operands" form. The explicit-operands form (specified with the LODS mnemonic) allows the source operand to be specified explicitly. Here, the source operand should be a symbol that indicates the size and location of the source value. The destination operand is then automatically selected to match the size of the source operand (the AL register for byte operands, AX for word operands, and EAX for doubleword operands). This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the DS:(E)SI registers, which must be loaded correctly before the load string instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the LODS instructions. Here also DS:(E)SI is assumed to be the source operand and the AL, AX, or EAX register is assumed to be the destination operand. The size of the source and destination operands is selected with the mnemonic: LODSB (byte loaded into register AL), LODSW (word loaded into AX), or LODSD (doubleword loaded into EAX).

After the byte, word, or doubleword is transferred from the memory location into the AL, AX, or EAX register, the (E)SI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI register is incremented; if the DF flag is 1, the ESI register is decremented.) The (E)SI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

In 64-bit mode, use of the REX.W prefix promotes operation to 64 bits. LODS/LODSQ load the quadword at address (R)SI into RAX. The (R)SI register is then incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register.

The LODS, LODSB, LODSW, and LODSD instructions can be preceded by the REP prefix for block loads of ECX bytes, words, or doublewords. More often, however, these instructions are used within a LOOP construct because further processing of the data moved into the register is usually necessary before the next transfer can be made. See "REP—Repeat String Operation (Prefix)" in Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B, for a description of the REP prefix.

Operation

```
IF AL := SRC; (* Byte load *)
    THEN AL := SRC; (* Byte load *)
        IF DF = 0
            THEN (E)SI := (E)SI + 1;
            ELSE (E)SI := (E)SI – 1;
        FI;
ELSE IF AX := SRC; (* Word load *)
    THEN IF DF = 0
            THEN (E)SI := (E)SI + 2;
            ELSE (E)SI := (E)SI – 2;
        IF;
    FI;
ELSE IF EAX := SRC; (* Doubleword load *)
    THEN IF DF = 0
            THEN (E)SI := (E)SI + 4;
            ELSE (E)SI := (E)SI – 4;
        FI;
    FI;
ELSE IF RAX := SRC; (* Quadword load *)
    THEN IF DF = 0
            THEN (R)SI := (R)SI + 8;
            ELSE (R)SI := (R)SI – 8;
        FI;
    FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

LOOP/LOOP*cc*—Loop According to ECX Counter

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
E2 cb	LOOP rel8	D	Valid	Valid	Decrement count; jump short if count ≠ 0.
E1 cb	LOOPE rel8	D	Valid	Valid	Decrement count; jump short if count ≠ 0 and ZF = 1.
E0 cb	LOOPNE rel8	D	Valid	Valid	Decrement count; jump short if count ≠ 0 and ZF = 0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
D	Offset	N/A	N/A	N/A

Description

Performs a loop operation using the RCX, ECX or CX register as a counter (depending on whether address size is 64 bits, 32 bits, or 16 bits). Note that the LOOP instruction ignores REX.W; but 64-bit address size can be over-ridden using a 67H prefix.

Each time the LOOP instruction is executed, the count register is decremented, then checked for 0. If the count is 0, the loop is terminated and program execution continues with the instruction following the LOOP instruction. If the count is not zero, a near jump is performed to the destination (target) operand, which is presumably the instruction at the beginning of the loop.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the IP/EIP/RIP register). This offset is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit immediate value, which is added to the instruction pointer. Offsets of −128 to +127 are allowed with this instruction.

Some forms of the loop instruction (LOOP*cc*) also accept the ZF flag as a condition for terminating the loop before the count reaches zero. With these forms of the instruction, a condition code (*cc*) is associated with each instruction to indicate the condition being tested for. Here, the LOOP*cc* instruction itself does not affect the state of the ZF flag; the ZF flag is changed by other instructions in the loop.

Operation

```
IF (AddressSize = 32)
    THEN Count is ECX;
ELSE IF (AddressSize = 64)
    Count is RCX;
ELSE Count is CX;
FI;

Count := Count – 1;

IF Instruction is not LOOP
    THEN
        IF (Instruction := LOOPE) or (Instruction := LOOPZ)
            THEN IF (ZF = 1) and (Count ≠ 0)
                    THEN BranchCond := 1;
                    ELSE BranchCond := 0;
            FI;
            ELSE (Instruction = LOOPNE) or (Instruction = LOOPNZ)
                IF (ZF = 0 ) and (Count ≠ 0)
                    THEN BranchCond := 1;
                    ELSE BranchCond := 0;
                FI;
```

```
        FI;
    ELSE (* Instruction = LOOP *)
        IF (Count ≠ 0)
            THEN BranchCond := 1;
            ELSE BranchCond := 0;
        FI;
FI;

IF BranchCond = 1
    THEN
        IF in 64-bit mode (* OperandSize = 64 *)
            THEN
                tempRIP := RIP + SignExtend(DEST);
                IF tempRIP is not canonical
                    THEN #GP(0);
                ELSE RIP := tempRIP;
                FI;
            ELSE
                tempEIP := EIP   SignExtend(DEST);
                IF OperandSize   16
                    THEN tempEIP := tempEIP AND 0000FFFFH;
                FI;
                IF tempEIP is not within code segment limit
                    THEN #GP(0);
                    ELSE EIP := tempEIP;
                FI;
        FI;
    ELSE
        Terminate loop and continue program execution at (R/E)IP;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the offset being jumped to is beyond the limits of the CS segment.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If the offset being jumped to is beyond the limits of the CS segment or is outside of the effective address space from 0 to FFFFH. This condition can occur if a 32-bit address size override prefix is used.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the offset being jumped to is in a non-canonical form.
#UD	If the LOCK prefix is used.

LSL—Load Segment Limit

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 03 /r	LSL r16, r16/m16	RM	Valid	Valid	Load segment limit from specified descriptor.
0F 03 /r	LSL r32, r32/m16[1]	RM	Valid	Valid	Load segment limit from specified descriptor.
REX.W + 0F 03 /r	LSL r64, r32/m16[1]	RM	Valid	Valid	Load segment limit from specified descriptor.

NOTES:

1. Regardless of operand size, only bits 15:0 of a register operand are used. Other bits are ignored.

Instruction Operand Encoding

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

Description

Loads the segment limit from the segment descriptor (see below) specified with the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the EFLAGS register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. If the source operand is a memory address, only 16 bits of data are accessed. The destination operand is a general-purpose register.

The processor performs access checks as part of the loading process. Once loaded in the destination register, software can compare the segment limit with the offset of a pointer.

The segment limit is a 20-bit value contained in bytes 0 and 1 and in the first 4 bits of byte 6 of the segment descriptor. If the descriptor has a byte granular segment limit (the granularity flag is set to 0), the destination operand is loaded with a byte granular value (byte limit) as read from the descriptor. If the descriptor has a page granular segment limit (the granularity flag is set to 1), the LSL instruction will translate the page granular limit (page limit) into a byte limit before loading it into the destination operand. The translation is performed by shifting the 20-bit "raw" limit left 12 bits and filling the low-order 12 bits with 1s.

When the operand size is 16 bits, a valid 32-bit byte limit is computed; however, the upper 16 bits are truncated and only the low-order 16 bits are loaded into the destination operand; the upper bits of the destination are unmodified. When the operand size is 32 bits, the 32-bit byte limit is loaded into the destination operand; the upper bits of the destination are cleared. When the operand is 64 bits, the 32-bit byte limit is zero-extended to 64 bits and loaded into the destination operand. (The behavior with 32-bit and 64-bit operand sizes is identical.)

This instruction performs the following checks before it loads the segment limit into the destination register:

- Checks that the segment selector is not NULL.
- Checks that the segment selector points to a descriptor that is within the limits of the GDT or LDT being accessed
- Checks that the descriptor type is valid for this instruction. All code and data segment descriptors are valid for (can be accessed with) the LSL instruction. The valid special segment and gate descriptor types are given in the Table 3-59.
- If the segment is not a conforming code segment, the instruction checks that the specified segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL of the segment selector).

If the segment descriptor cannot be accessed or is an invalid type for the instruction, the ZF flag is cleared and no value is loaded in the destination operand.

Table 3-59. Segment and Gate Descriptor Types

Type	Protected Mode		IA-32e Mode	
	Name	Valid	Name	Valid
0	Reserved	No	Reserved	No
1	Available 16-bit TSS	Yes	Reserved	No
2	LDT	Yes	LDT[1]	Yes
3	Busy 16-bit TSS	Yes	Reserved	No
4	16-bit call gate	No	Reserved	No
5	16-bit/32-bit task gate	No	Reserved	No
6	16-bit interrupt gate	No	Reserved	No
7	16-bit trap gate	No	Reserved	No
8	Reserved	No	Reserved	No
9	Available 32-bit TSS	Yes	64-bit TSS[1]	Yes
A	Reserved	No	Reserved	No
B	Busy 32-bit TSS	Yes	Busy 64-bit TSS[1]	Yes
C	32-bit call gate	No	64-bit call gate	No
D	Reserved	No	Reserved	No
E	32-bit interrupt gate	No	64-bit interrupt gate	No
F	32-bit trap gate	No	64-bit trap gate	No

NOTES:

1. In this case, the descriptor comprises 16 bytes; bits 12:8 of the upper 4 bytes must be 0.

Operation

```
IF SRC(Offset) > descriptor table limit
    THEN ZF := 0; FI;

Read segment descriptor;

IF SegmentDescriptor(Type) ≠ conforming code segment
and (CPL > DPL) OR (RPL > DPL)
or Segment type is not valid for instruction
        THEN
            ZF := 0;
        ELSE
            temp := SegmentLimit([SRC]);
            IF (SegmentDescriptor(G) = 1)
                THEN temp := (temp << 12) OR 00000FFFH;
            ELSE IF OperandSize = 32
                THEN DEST := temp; FI;
            ELSE IF OperandSize = 64 (* REX.W used *)
                THEN DEST := temp(* Zero-extended *); FI;
            ELSE (* OperandSize = 16 *)
                DEST := temp AND FFFFH;
            FI;
FI;
```

Flags Affected

The ZF flag is set to 1 if the segment limit is loaded successfully; otherwise, it is set to 0. The CF, OF, SF, AF, and PF flags are not modified.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#UD	The LSL instruction cannot be executed in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The LSL instruction cannot be executed in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

LTR—Load Task Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 00 /3	LTR r/m16	M	Valid	Valid	Load TR using segment selector in r/m16.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

Description

Loads the source operand into the segment selector field of the task register (TR). The source operand (a general-purpose register or a memory location) contains a segment selector that references a TSS descriptor in the global descriptor table (GDT) that in turn references a task state segment (TSS). The instruction loads the segment limit and base address for the TSS from the TSS descriptor. The TSS is marked busy, but a switch to the corresponding task does not occur.

If bits 15:2 of the source operand are 0, a general protection exception (#GP) occurs.

The operand-size attribute has no effect on this instruction.

The LTR instruction is provided for use in operating-system software; it cannot be used in application programs. It can be executed only when the CPL is 0. It is commonly used in initialization code to establish the first task to be executed. This instruction cannot be executed in real-address mode or in virtual-8086 mode.

In 64-bit mode, the operand size is still fixed at 16 bits. The instruction uses a 16-byte TSS descriptor that contains a 64-bit base address.

Operation

```
IF SRC is a NULL selector
    THEN #GP(0);
ELSE IF SRC(type) ≠ global OR SRC(Offset) > GDTR.limit
    THEN #GP(SRC);
ELSE
    Read SegmentDescriptor from GDT;
    IF SegmentDescriptor(Type) ≠ 9                          // available 32-bit TSS
        AND (IA32_EFER.LMA = 1 OR SegmentDescriptor(Type) ≠ 1) // available 16-bit TSS
            THEN #GP(segment selector); FI;
    IF segment descriptor is not present
        THEN #NP(segment selector); FI;
    SegmentDescriptor(busy) := 1;
    (* Locked read-modify-write operation on the entire descriptor when setting busy flag *)
    TaskRegister(SegmentSelector) := SRC;
    TaskRegister(SegmentDescriptor) := SegmentDescriptor;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the source operand contains a NULL segment selector.
	If the DS, ES, FS, or GS register is used to access memory and contains a NULL segment selector.
#GP(selector)	If the source selector points to a segment that is not an available TSS.
	If the selector points to the LDT or is beyond the GDT limit.
#NP(selector)	If the TSS descriptor is marked not present.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	The LTR instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The LTR instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode, as well as the following:

#GP(selector)	If the source selector points to a 16-bit TSS.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the current privilege level is not 0.
	If the memory address is in a non-canonical form.
	If the source operand contains a NULL segment selector.
#GP(selector)	If the source selector points to a segment that is not a TSS, to a 16-bit TSS, or to a TSS for a task that is already busy.
	If the source selector points to the LDT or is beyond the GDT limit.
	If the descriptor type of the upper 8-byte of the 16-byte descriptor is non-zero.
#NP(selector)	If the TSS descriptor is marked not present.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

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	RM	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	RM	V/V	LZCNT	Count the number of leading zero bits in r/m32, return result in r32.
F3 REX.W 0F BD /r LZCNT r64, r/m64	RM	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
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

LZCNT counts the number of leading most significant zero bits in a source operand (second operand) and returns the result in the destination (first operand). LZCNT is an extension of the BSR instruction. The key difference between the LZCNT and BSR instructions is that when the source operand is zero, LZCNT outputs the operand size to the destination operand, whereas BSR leaves the destination operand unmodified.

On processors that do not support LZCNT, the instruction byte encoding is executed as BSR.

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);

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	For an illegal address in the SS segment.
#UD	If LOCK prefix is used.

Virtual 8086 Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

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

CHAPTER 4
INSTRUCTION SET REFERENCE, M-U

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

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

4.1.1 General Description

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

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

The imm8 control byte for PCMPESTRM/PCMPESTRI/PCMPISTRM/PCMPISTRI encodes a significant amount of programmable control over the functionality of those instructions. Some functionality is unique to each instruction while some is common across some or all of the four instructions. This section describes functionality which is common across the four instructions.

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

PCMPxSTRx instructions perform arithmetic comparisons between all possible pairs of bytes or words, one from each packed input source operand. The boolean results of those comparisons are then aggregated in order to produce meaningful results. The imm8 control byte is used to affect the interpretation of individual input elements as well as control the arithmetic comparisons used and the specific aggregation scheme.

Specifically, the imm8 Control Byte consists of bit fields that control the following attributes:

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

4.1.2 Source Data Format

Table 4-1. Source Data Format

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

If the imm8 control byte has bit[0] cleared, each source contains 16 packed bytes. If the bit is set each source contains 8 packed words. If the imm8 control byte has bit[1] cleared, each input contains unsigned data. If the bit is set each source contains signed data.

4.1.3 Aggregation Operation

Table 4-2. Aggregation Operation

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

All 256 (64) possible comparisons are always performed. The individual Boolean results of those comparisons are referred by "BoolRes[*Reg/Mem element index, Reg element index*]." Comparisons evaluating to "True" are represented with a 1, False with a 0 (positive logic). The initial results are then aggregated into a 16-bit (8-bit) intermediate result (IntRes1) using one of the modes described in the table below, as determined by imm8 control byte bits[3:2].

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

Table 4-3. Aggregation Operation

Mode	Pseudocode

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

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

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

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

4.1.4 Polarity

IntRes1 may then be further modified by performing a 1's complement, according to the value of the imm8 control byte bit[4]. Optionally, a mask may be used such that only those IntRes1 bits which correspond to "valid" reg/mem input elements are complemented (note that the definition of a valid input element is dependent on the specific opcode and is defined in each opcode's description). The result of the possible negation is referred to as IntRes2.

Table 4-4. Polarity

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

4.1.5 Output Selection

Table 4-5. Output Selection

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

For PCMPESTRI/PCMPISTRI, the imm8 control byte bit[6] is used to determine if the index is of the least significant or most significant bit of IntRes2.

Table 4-6. Output Selection

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

Specifically for PCMPESTRM/PCMPISTRM, the imm8 control byte bit[6] is used to determine if the mask is a 16 (8) bit mask or a 128 bit byte/word mask.

4.1.6 Valid/Invalid Override of Comparisons

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

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

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

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

4.1.7 Summary of Imm8 Control byte

Table 4-8. Summary of Imm8 Control Byte

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

4.1.8 Diagram Comparison and Aggregation Process

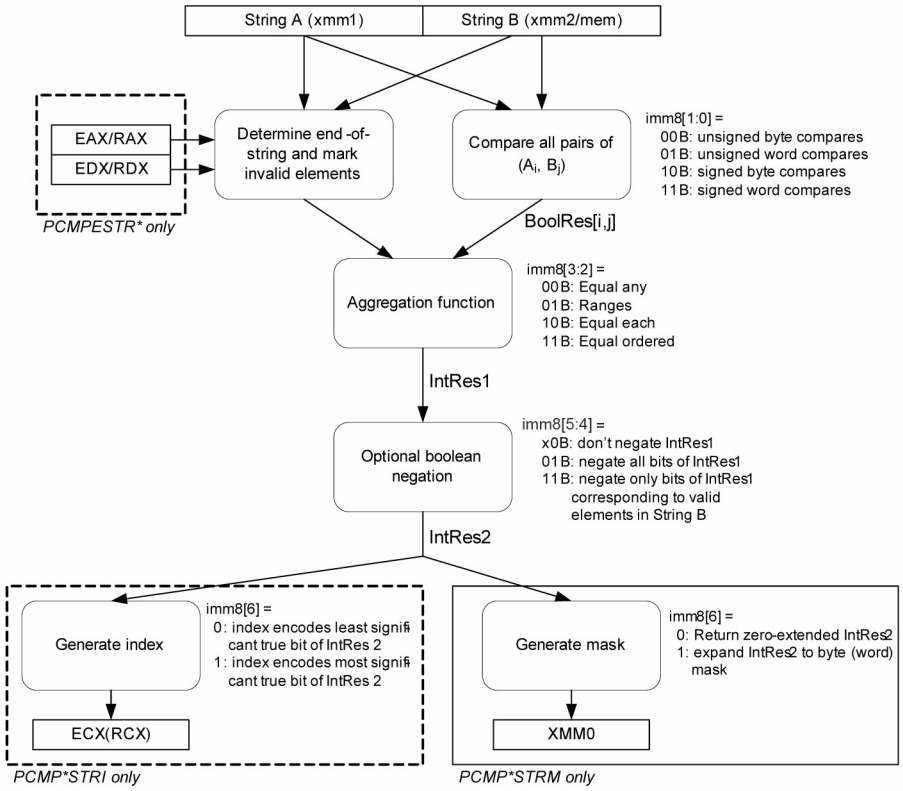


Figure 4-1. Operation of PCMPSTRx and PCMPESTRx

4.2 COMMON TRANSFORMATION AND PRIMITIVE FUNCTIONS FOR SHA1XXX AND SHA256XXX

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

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

  f0(B,C,D) := (B AND C) XOR ((NOT(B) AND D)

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

  f1(B,C,D) := B XOR C XOR D

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

  f2(B,C,D) := (B AND C) XOR (B AND D) XOR (C AND D)

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

  f3(B,C,D) := B XOR C XOR D

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

Ch(E,F,G) := (E AND F) XOR ((NOT E) AND G)

- Maj(): A bit oriented logical operation that derives a new dword from three SHA256 state variables (dword).
  Maj(A,B,C) := (A AND B) XOR (A AND C) XOR (B AND C)

ROR is rotate right operation
(A ROR N) := A[N-1:0] || A[Width-1:N]

ROL is rotate left operation
(A ROL N) := A ROR (Width-N)

SHR is the right shift operation
(A SHR N) := ZEROES[N-1:0] || A[Width-1:N]

- Σ_0( ): A bit oriented logical and rotational transformation performed on a dword SHA256 state variable.
  Σ_0(A) := (A ROR 2) XOR (A ROR 13) XOR (A ROR 22)

- Σ_1( ): A bit oriented logical and rotational transformation performed on a dword SHA256 state variable.
  Σ_1(E) := (E ROR 6) XOR (E ROR 11) XOR (E ROR 25)

- σ_0( ): A bit oriented logical and rotational transformation performed on a SHA256 message dword used in the message scheduling.
  σ_0(W) := (W ROR 7) XOR (W ROR 18) XOR (W SHR 3)

- σ_1( ): A bit oriented logical and rotational transformation performed on a SHA256 message dword used in the message scheduling.
  σ_1(W) := (W ROR 17) XOR (W ROR 19) XOR (W SHR 10)

- K_i: SHA1 Constants dependent on immediate i.
  K0 = 0x5A827999
  K1 = 0x6ED9EBA1
  K2 = 0X8F1BBCDC
  K3 = 0xCA62C1D6

4.3 INSTRUCTIONS (M-U)

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

MASKMOVDQU—Store Selected Bytes of Double Quadword

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

Instruction Operand Encoding[1]

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

Description

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

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

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

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

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

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

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

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

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

1. ModRM.MOD = 011B required

Operation

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

Intel C/C++ Compiler Intrinsic Equivalent

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

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions," additionally:

#UD	If VEX.L= 1
	If VEX.vvvv ≠ 1111B.

MASKMOVQ—Store Selected Bytes of Quadword

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

Instruction Operand Encoding

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

Description

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

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

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

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

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

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

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

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

Operation

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

Intel C/C++ Compiler Intrinsic Equivalent

void _mm_maskmove_si64(__m64d, __m64n, char * p)

Other Exceptions

See Table 25-8, "Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

MAXPD—Maximum of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5F /r MAXPD xmm1, xmm2/m128	A	V/V	SSE2	Return the maximum double precision floating-point values between xmm1 and xmm2/m128.
VEX.128.66.0F.WIG 5F /r VMAXPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the maximum double precision floating-point values between xmm2 and xmm3/m128.
VEX.256.66.0F.WIG 5F /r VMAXPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the maximum packed double precision floating-point values between ymm2 and ymm3/m256.
EVEX.128.66.0F.W1 5F /r VMAXPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Return the maximum packed double precision floating-point values between xmm2 and xmm3/m128/m64bcst and store result in xmm1 subject to writemask k1.
EVEX.256.66.0F.W1 5F /r VMAXPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Return the maximum packed double precision floating-point values between ymm2 and ymm3/m256/m64bcst and store result in ymm1 subject to writemask k1.
EVEX.512.66.0F.W1 5F /r VMAXPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}	C	V/V	AVX512F OR AVX10.1	Return the maximum packed double precision floating-point values between zmm2 and zmm3/m512/m64bcst and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

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

Description

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

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

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

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

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

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

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

Operation

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

VMAXPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := MAX(SRC1[i+63:i], SRC2[63:0])
                ELSE
                    DEST[i+63:i] := MAX(SRC1[i+63:i], SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] := 0             ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMAXPD (VEX.256 Encoded Version)

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

VMAXPD (VEX.128 Encoded Version)

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

MAXPD (128-bit Legacy SSE Version)

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

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-48, “Type E2 Class Exception Conditions.”

MAXPS—Maximum of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5F /r MAXPS xmm1, xmm2/m128	A	V/V	SSE	Return the maximum single precision floating-point values between xmm1 and xmm2/mem.
VEX.128.0F.WIG 5F /r VMAXPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the maximum single precision floating-point values between xmm2 and xmm3/mem.
VEX.256.0F.WIG 5F /r VMAXPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the maximum single precision floating-point values between ymm2 and ymm3/mem.
EVEX.128.0F.W0 5F /r VMAXPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Return the maximum packed single precision floating-point values between xmm2 and xmm3/m128/m32bcst and store result in xmm1 subject to writemask k1.
EVEX.256.0F.W0 5F /r VMAXPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Return the maximum packed single precision floating-point values between ymm2 and ymm3/m256/m32bcst and store result in ymm1 subject to writemask k1.
EVEX.512.0F.W0 5F /r VMAXPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}	C	V/V	AVX512F OR AVX10.1	Return the maximum packed single precision floating-point values between zmm2 and zmm3/m512/m32bcst and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

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

Description

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

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

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

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

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

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

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

Operation

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

VMAXPS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] := MAX(SRC1[i+31:i], SRC2[31:0])
                ELSE
                    DEST[i+31:i] := MAX(SRC1[i+31:i], SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] := 0             ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMAXPS (VEX.256 Encoded Version)

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

VMAXPS (VEX.128 Encoded Version)
DEST[31:0] := MAX(SRC1[31:0], SRC2[31:0])
DEST[63:32] := MAX(SRC1[63:32], SRC2[63:32])
DEST[95:64] := MAX(SRC1[95:64], SRC2[95:64])
DEST[127:96] := MAX(SRC1[127:96], SRC2[127:96])
DEST[MAXVL-1:128] := 0

MAXPS (128-bit Legacy SSE Version)
DEST[31:0] := MAX(DEST[31:0], SRC[31:0])
DEST[63:32] := MAX(DEST[63:32], SRC[63:32])
DEST[95:64] := MAX(DEST[95:64], SRC[95:64])
DEST[127:96] := MAX(DEST[127:96], SRC[127:96])
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-48, “Type E2 Class Exception Conditions.”

MAXSD—Return Maximum Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5F /r MAXSD xmm1, xmm2/m64	A	V/V	SSE2	Return the maximum scalar double precision floating-point value between xmm2/m64 and xmm1.
VEX.LIG.F2.0F.WIG 5F /r VMAXSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Return the maximum scalar double precision floating-point value between xmm3/m64 and xmm2.
EVEX.LLIG.F2.0F.W1 5F /r VMAXSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	C	V/V	AVX512F OR AVX10.1	Return the maximum scalar double precision floating-point value between xmm3/m64 and xmm2.

Instruction Operand Encoding

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

Description

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

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

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

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

VEX.128 and EVEX encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the write-mask.

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

Operation

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

VMAXSD (EVEX Encoded Version)

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

VMAXSD (VEX.128 Encoded Version)

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

MAXSD (128-bit Legacy SSE Version)

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

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

Invalid (Including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-20, "Type 3 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-49, "Type E3 Class Exception Conditions."

MAXSS—Return Maximum Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5F /r MAXSS xmm1, xmm2/m32	A	V/V	SSE	Return the maximum scalar single precision floating-point value between xmm2/m32 and xmm1.
VEX.LIG.F3.0F.WIG 5F /r VMAXSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Return the maximum scalar single precision floating-point value between xmm3/m32 and xmm2.
EVEX.LLIG.F3.0F.W0 5F /r VMAXSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	C	V/V	AVX512F OR AVX10.1	Return the maximum scalar single precision floating-point value between xmm3/m32 and xmm2.

Instruction Operand Encoding

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

Description

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

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

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

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

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

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

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

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

Operation

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

VMAXSS (EVEX Encoded Version)

```
IF k1[0] or *no writemask*
    THEN      DEST[31:0] := MAX(SRC1[31:0], SRC2[31:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VMAXSS (VEX.128 Encoded Version)

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

MAXSS (128-bit Legacy SSE Version)

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

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

Invalid (Including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-20, "Type 3 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-49, "Type E3 Class Exception Conditions."

MFENCE—Memory Fence

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F AE F0 MFENCE	ZO	V/V	SSE2	Serializes load and store operations.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

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

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

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

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

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

Operation

```
Wait_On_Following_Loads_And_Stores_Until(preceding_loads_and_stores_globally_visible);
```

Intel C/C++ Compiler Intrinsic Equivalent

void _mm_mfence(void)

Exceptions (All Modes of Operation)

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

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

MINPD—Minimum of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5D /r MINPD xmm1, xmm2/m128	A	V/V	SSE2	Return the minimum double precision floating-point values between xmm1 and xmm2/mem
VEX.128.66.0F.WIG 5D /r VMINPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the minimum double precision floating-point values between xmm2 and xmm3/mem.
VEX.256.66.0F.WIG 5D /r VMINPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the minimum packed double precision floating-point values between ymm2 and ymm3/mem.
EVEX.128.66.0F.W1 5D /r VMINPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Return the minimum packed double precision floating-point values between xmm2 and xmm3/m128/m64bcst and store result in xmm1 subject to writemask k1.
EVEX.256.66.0F.W1 5D /r VMINPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Return the minimum packed double precision floating-point values between ymm2 and ymm3/m256/m64bcst and store result in ymm1 subject to writemask k1.
EVEX.512.66.0F.W1 5D /r VMINPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}	C	V/V	AVX512F OR AVX10.1	Return the minimum packed double precision floating-point values between zmm2 and zmm3/m512/m64bcst and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

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

Description

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

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

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

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

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

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

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

Operation

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

VMINPD (EVEX Encoded Version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := MIN(SRC1[i+63:i], SRC2[63:0])
                ELSE
                    DEST[i+63:i] := MIN(SRC1[i+63:i], SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] := 0             ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMINPD (VEX.256 Encoded Version)

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

VMINPD (VEX.128 Encoded Version)

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

MINPD (128-bit Legacy SSE Version)

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

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, "Type 2 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-48, "Type E2 Class Exception Conditions."

MINPS—Minimum of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5D /r MINPS xmm1, xmm2/m128	A	V/V	SSE	Return the minimum single precision floating-point values between xmm1 and xmm2/mem.
VEX.128.0F.WIG 5D /r VMINPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the minimum single precision floating-point values between xmm2 and xmm3/mem.
VEX.256.0F.WIG 5D /r VMINPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the minimum single double precision floating-point values between ymm2 and ymm3/mem.
EVEX.128.0F.W0 5D /r VMINPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Return the minimum packed single precision floating-point values between xmm2 and xmm3/m128/m32bcst and store result in xmm1 subject to writemask k1.
EVEX.256.0F.W0 5D /r VMINPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Return the minimum packed single precision floating-point values between ymm2 and ymm3/m256/m32bcst and store result in ymm1 subject to writemask k1.
EVEX.512.0F.W0 5D /r VMINPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}	C	V/V	AVX512F OR AVX10.1	Return the minimum packed single precision floating-point values between zmm2 and zmm3/m512/m32bcst and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

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

Description

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

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

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

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

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

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

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

Operation

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

VMINPS (EVEX Encoded Version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] := MIN(SRC1[i+31:i], SRC2[31:0])
                ELSE
                    DEST[i+31:i] := MIN(SRC1[i+31:i], SRC2[i+31:i])
            FI;
            ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] := 0             ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMINPS (VEX.256 Encoded Version)

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

VMINPS (VEX.128 Encoded Version)
DEST[31:0] := MIN(SRC1[31:0], SRC2[31:0])
DEST[63:32] := MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] := MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] := MIN(SRC1[127:96], SRC2[127:96])
DEST[MAXVL-1:128] := 0

MINPS (128-bit Legacy SSE Version)
DEST[31:0] := MIN(SRC1[31:0], SRC2[31:0])
DEST[63:32] := MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] := MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] := MIN(SRC1[127:96], SRC2[127:96])
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, "Type 2 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-48, "Type E2 Class Exception Conditions."

MINSD—Return Minimum Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5D /r MINSD xmm1, xmm2/m64	A	V/V	SSE2	Return the minimum scalar double precision floating-point value between xmm2/m64 and xmm1.
VEX.LIG.F2.0F.WIG 5D /r VMINSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Return the minimum scalar double precision floating-point value between xmm3/m64 and xmm2.
EVEX.LLIG.F2.0F.W1 5D /r VMINSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	C	V/V	AVX512F OR AVX10.1	Return the minimum scalar double precision floating-point value between xmm3/m64 and xmm2.

Instruction Operand Encoding

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

Description

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

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

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

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

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

VEX.128 and EVEX encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the write-mask.

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

Operation

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

MINSD (EVEX Encoded Version)

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

MINSD (VEX.128 Encoded Version)

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

MINSD (128-bit Legacy SSE Version)

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

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-20, "Type 3 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-49, "Type E3 Class Exception Conditions."

MINSS—Return Minimum Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5D /r MINSS xmm1,xmm2/m32	A	V/V	SSE	Return the minimum scalar single precision floating-point value between xmm2/m32 and xmm1.
VEX.LIG.F3.0F.WIG 5D /r VMINSS xmm1,xmm2, xmm3/m32	B	V/V	AVX	Return the minimum scalar single precision floating-point value between xmm3/m32 and xmm2.
EVEX.LLIG.F3.0F.W0 5D /r VMINSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	C	V/V	AVX512F OR AVX10.1	Return the minimum scalar single precision floating-point value between xmm3/m32 and xmm2.

Instruction Operand Encoding

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

Description

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

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

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

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

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

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

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

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

Operation

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

MINSS (EVEX Encoded Version)

```
IF k1[0] or *no writemask*
    THEN      DEST[31:0] := MIN(SRC1[31:0], SRC2[31:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VMINSS (VEX.128 Encoded Version)

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

MINSS (128-bit Legacy SSE Version)

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

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

Invalid (Including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, "Type 2 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-48, "Type E2 Class Exception Conditions."

MONITOR—Set Up Monitor Address

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

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

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

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

The address range must use memory of the write-back type. Only write-back memory will correctly trigger the monitoring hardware. Additional information on determining what address range to use in order to prevent false wake-ups is described in Chapter 11, "Multiple-Processor Management," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

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

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

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

Operation

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

Intel C/C++ Compiler Intrinsic Equivalent

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

Numeric Exceptions

None.

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

MOV—Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
88 /r	MOV r/m8[1], r8[1]	MR	Valid	Valid	Move r8 to r/m8.
89 /r	MOV r/m16, r16	MR	Valid	Valid	Move r16 to r/m16.
89 /r	MOV r/m32, r32	MR	Valid	Valid	Move r32 to r/m32.
REX.W + 89 /r	MOV r/m64, r64	MR	Valid	N.E.	Move r64 to r/m64.
8A /r	MOV r8[1], r/m8[1]	RM	Valid	Valid	Move r/m8 to r8.
8B /r	MOV r16, r/m16	RM	Valid	Valid	Move r/m16 to r16.
8B /r	MOV r32, r/m32	RM	Valid	Valid	Move r/m32 to r32.
REX.W + 8B /r	MOV r64, r/m64	RM	Valid	N.E.	Move r/m64 to r64.
8C /r	MOV r/m16, Sreg[2]	MR	Valid	Valid	Move segment register to r/m16.
8C /r	MOV r16/r32/m16, Sreg[2]	MR	Valid	Valid	Move zero extended 16-bit segment register to r16/r32/m16.
REX.W + 8C /r	MOV r64/m16, Sreg[2]	MR	Valid	Valid	Move zero extended 16-bit segment register to r64/m16.
8E /r	MOV Sreg, r/m16[2]	RM	Valid	Valid	Move r/m16 to segment register.
REX.W + 8E /r	MOV Sreg, r/m64[2]	RM	Valid	Valid	Move lower 16 bits of r/m64 to segment register.
A0	MOV AL, moffs8[3]	FD	Valid	Valid	Move byte at (seg:offset) to AL.
REX.W + A0	MOV AL, moffs8[3]	FD	Valid	N.E.	Move byte at (offset) to AL.
A1	MOV AX, moffs16[3]	FD	Valid	Valid	Move word at (seg:offset) to AX.
A1	MOV EAX, moffs32[3]	FD	Valid	Valid	Move doubleword at (seg:offset) to EAX.
REX.W + A1	MOV RAX, moffs64[3]	FD	Valid	N.E.	Move quadword at (offset) to RAX.
A2	MOV moffs8, AL	TD	Valid	Valid	Move AL to (seg:offset).
REX.W + A2	MOV moffs8[1], AL	TD	Valid	N.E.	Move AL to (offset).
A3	MOV moffs16[3], AX	TD	Valid	Valid	Move AX to (seg:offset).
A3	MOV moffs32[3], EAX	TD	Valid	Valid	Move EAX to (seg:offset).
REX.W + A3	MOV moffs64[3], RAX	TD	Valid	N.E.	Move RAX to (offset).
B0+ rb ib	MOV r8[1], imm8	OI	Valid	Valid	Move imm8 to r8.
B8+ rw iw	MOV r16, imm16	OI	Valid	Valid	Move imm16 to r16.
B8+ rd id	MOV r32, imm32	OI	Valid	Valid	Move imm32 to r32.
REX.W + B8+ rd io	MOV r64, imm64	OI	Valid	N.E.	Move imm64 to r64.
C6 /0 ib	MOV r/m8[1], imm8	MI	Valid	Valid	Move imm8 to r/m8.
C7 /0 iw	MOV r/m16, imm16	MI	Valid	Valid	Move imm16 to r/m16.
C7 /0 id	MOV r/m32, imm32	MI	Valid	Valid	Move imm32 to r/m32.
REX.W + C7 /0 id	MOV r/m64, imm32	MI	Valid	N.E.	Move imm32 sign extended to 64-bits to r/m64.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.
2. In 32-bit mode, the assembler may insert the 16-bit operand-size prefix with this instruction (see the following "Description" section for further information).
3. The moffs8, moffs16, moffs32, and moffs64 operands specify a simple offset relative to the segment base, where 8, 16, 32, and 64 refer to the size of the data. The address-size attribute of the instruction determines the size of the offset, either 16, 32, or 64 bits.

Instruction Operand Encoding

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

Description

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

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

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

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

Loading the SS register with a MOV instruction suppresses or inhibits some debug exceptions and inhibits interrupts on the following instruction boundary. (The inhibition ends after delivery of an exception or the execution of the next instruction.) This behavior allows a stack pointer to be loaded into the ESP register with the next instruction (MOV ESP, **stack-pointer value**) before an event can be delivered. See Section 7.8.3, "Masking Exceptions and Interrupts When Switching Stacks," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A. Intel recommends that software use the LSS instruction to load the SS register and ESP together.

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

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

Operation

```
DEST := SRC;
```

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

```
IF SS is loaded
    THEN
        IF segment selector is NULL
            THEN #GP(0); FI;
        IF segment selector index is outside descriptor table limits
        OR segment selector's RPL ≠ CPL
```

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

Flags Affected

None.

Protected Mode Exceptions

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

#UD	If attempt is made to load the CS register. If the LOCK prefix is used.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

MOV—Move to/from Control Registers

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

NOTES:

1. MOV CR* instructions, except for MOV CR8, are serializing instructions. MOV CR8 is not architecturally defined as a serializing instruction. For more information, see Chapter 11 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

Instruction Operand Encoding

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

Description

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

At the opcode level, the *reg* field within the ModR/M byte specifies which of the control registers is loaded or read. The 2 bits in the *mod* field are ignored. The *r/m* field specifies the general-purpose register loaded or read. Some of the bits in CR0, CR3, and CR4 are reserved and must be written with zeros. Attempting to set any reserved bits in CR0[31:0] is ignored. Attempting to set any reserved bits in CR0[63:32] results in a general-protection exception, #GP(0). When PCIDs are not enabled, bits 2:0 and bits 11:5 of CR3 are not used and attempts to set them are ignored. See the next paragraph for treatment of bits reserved in CR3. Attempting to set any reserved bits in CR4 results in #GP(0). On Pentium 4, Intel Xeon and P6 family processors, CR0.ET remains set after any load of CR0; attempts to clear this bit have no impact.

Normally, MAXPHYADDR is the value enumerated in CPUID.80000008H:EAX[7:0]. However, if IA32_TME_ACTIVATE[0] = 1 (indicating that TME has been configured), MAXPHYADDR is reduced by the value of IA32_TME_ACTIVATE[39:36] when a logical processor is outside secure arbitration mode (SEAM; see Chapter 35 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Volume 3); the value is not reduced in SEAM. An attempt to set any reserved bit in CR3[63:MAXPHYADDR] results in #GP(0).

In certain cases, these instructions have the side effect of invalidating entries in the TLBs and the paging-structure caches. See Section 5.10.4.1, "Operations that Invalidate TLBs and Paging-Structure Caches," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, for details.

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

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

If CR4.PCIDE = 1, bit 63 of the source operand to MOV to CR3 determines whether the instruction invalidates entries in the TLBs and the paging-structure caches (see Section 5.10.4.1, "Operations that Invalidate TLBs and Paging-Structure Caches," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A). The instruction does not modify bit 63 of CR3, which is reserved and always 0.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 27 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C, for more information about the behavior of this instruction in VMX non-root operation.

Operation

```
DEST := SRC;
```

Flags Affected

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

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).
	If an attempt is made to write a 1 to any reserved bit in CR4.
	If an attempt is made to write 1 to CR4.PCIDE.
	If any of the reserved bits are set in the page-directory pointers table (PDPT) and the loading of a control register causes the PDPT to be loaded into the processor.
	If an attempt is made to activate IA-32e mode and either the current CS has the L-bit set or the TR references a 16-bit TSS.
#UD	If the LOCK prefix is used.
	If an attempt is made to access CR1, CR5, CR6, CR7, or CR9–CR15.

Real-Address Mode Exceptions

#GP	If an attempt is made to write a 1 to any reserved bit in CR4.
	If an attempt is made to write 1 to CR4.PCIDE.
	If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0).
	If an attempt is made to activate IA-32e mode and either the current CS has the L-bit set or the TR references a 16-bit TSS.
#UD	If the LOCK prefix is used.
	If an attempt is made to access CR1, CR5, CR6, CR7, or CR9–CR15.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).
	If an attempt is made to change CR4.PCIDE from 0 to 1 while CR3[11:0] ≠ 000H.
	If an attempt is made to clear CR0.PG[bit 31] while CR4.PCIDE = 1.
	If an attempt is made to leave IA-32e mode by clearing CR4.PAE[bit 5].
#UD	If the LOCK prefix is used.
	If an attempt is made to access CR1, CR5, CR6, CR7, or CR9–CR15.

64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).
	If an attempt is made to change CR4.PCIDE from 0 to 1 while CR3[11:0] ≠ 000H.
	If an attempt is made to clear CR0.PG[bit 31].
	If an attempt is made to write a 1 to any reserved bit in CR4.
	If an attempt is made to write a 1 to any reserved bit in CR8.
	If an attempt is made to write a 1 to any reserved bit in CR3[63:MAXPHYADDR].
	If an attempt is made to leave IA-32e mode by clearing CR4.PAE[bit 5].
#UD	If the LOCK prefix is used.
	If an attempt is made to access CR1, CR5, CR6, CR7, or CR9–CR15.
	If the REX.R prefix is used to specify a register other than CR8.

MOV—Move to/from Debug Registers

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

Instruction Operand Encoding

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

Description

Moves the contents of a debug register (DR0, DR1, DR2, DR3, DR4, DR5, DR6, or DR7) to a general-purpose register or vice versa. The operand size for these instructions is always 32 bits in non-64-bit modes, regardless of the operand-size attribute. (See Section 20.2, "Debug Registers", of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, for a detailed description of the flags and fields in the debug registers.)

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

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

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

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

Operation

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

Flags Affected

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

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

MOVAPD—Move Aligned Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 28 /r MOVAPD xmm1, xmm2/m128	A	V/V	SSE2	Move aligned packed double precision floating-point values from xmm2/mem to xmm1.
66 0F 29 /r MOVAPD xmm2/m128, xmm1	B	V/V	SSE2	Move aligned packed double precision floating-point values from xmm1 to xmm2/mem.
VEX.128.66.0F.WIG 28 /r VMOVAPD xmm1, xmm2/m128	A	V/V	AVX	Move aligned packed double precision floating-point values from xmm2/mem to xmm1.
VEX.128.66.0F.WIG 29 /r VMOVAPD xmm2/m128, xmm1	B	V/V	AVX	Move aligned packed double precision floating-point values from xmm1 to xmm2/mem.
VEX.256.66.0F.WIG 28 /r VMOVAPD ymm1, ymm2/m256	A	V/V	AVX	Move aligned packed double precision floating-point values from ymm2/mem to ymm1.
VEX.256.66.0F.WIG 29 /r VMOVAPD ymm2/m256, ymm1	B	V/V	AVX	Move aligned packed double precision floating-point values from ymm1 to ymm2/mem.
EVEX.128.66.0F.W1 28 /r VMOVAPD xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed double precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F.W1 28 /r VMOVAPD ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed double precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F.W1 28 /r VMOVAPD zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F OR AVX10.1	Move aligned packed double precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.66.0F.W1 29 /r VMOVAPD xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed double precision floating-point values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W1 29 /r VMOVAPD ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed double precision floating-point values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W1 29 /r VMOVAPD zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1	Move aligned packed double precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

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

Description

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

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit versions), 32-byte (256-bit version) or 64-byte (EVEX.512 encoded version) boundary or a general-protection

exception (#GP) will be generated. For EVEX encoded versions, the operand must be aligned to the size of the memory operand. To move double precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

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

EVEX.512 encoded version:

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

VEX.256 and EVEX.256 encoded versions:

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

128-bit versions:

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

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

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

Operation

VMOVAPD (EVEX Encoded Versions, Register-Copy Form)

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

VMOVAPD (EVEX Encoded Versions, Store-Form)

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

    FI;
ENDFOR;
```

VMOVAPD (EVEX Encoded Versions, Load-Form)

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

VMOVAPD (VEX.256 Encoded Version, Load - and Register Copy)

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

VMOVAPD (VEX.256 Encoded Version, Store-Form)

```
DEST[255:0] := SRC[255:0]
```

VMOVAPD (VEX.128 Encoded Version, Load - and Register Copy)

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

MOVAPD (128-bit Load- and Register-Copy- Form Legacy SSE Version)

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

(V)MOVAPD (128-bit Store-Form Version)

```
DEST[127:0] := SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2 in Table 2-18, "Type 1 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-46, "Type E1 Class Exception Conditions."
Additionally:

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

MOVAPS—Move Aligned Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 28 /r MOVAPS xmm1, xmm2/m128	A	V/V	SSE	Move aligned packed single precision floating-point values from xmm2/mem to xmm1.
NP 0F 29 /r MOVAPS xmm2/m128, xmm1	B	V/V	SSE	Move aligned packed single precision floating-point values from xmm1 to xmm2/mem.
VEX.128.0F.WIG 28 /r VMOVAPS xmm1, xmm2/m128	A	V/V	AVX	Move aligned packed single precision floating-point values from xmm2/mem to xmm1.
VEX.128.0F.WIG 29 /r VMOVAPS xmm2/m128, xmm1	B	V/V	AVX	Move aligned packed single precision floating-point values from xmm1 to xmm2/mem.
VEX.256.0F.WIG 28 /r VMOVAPS ymm1, ymm2/m256	A	V/V	AVX	Move aligned packed single precision floating-point values from ymm2/mem to ymm1.
VEX.256.0F.WIG 29 /r VMOVAPS ymm2/m256, ymm1	B	V/V	AVX	Move aligned packed single precision floating-point values from ymm1 to ymm2/mem.
EVEX.128.0F.W0 28 /r VMOVAPS xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed single precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.0F.W0 28 /r VMOVAPS ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed single precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.0F.W0 28 /r VMOVAPS zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F OR AVX10.1	Move aligned packed single precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.0F.W0 29 /r VMOVAPS xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed single precision floating-point values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.0F.W0 29 /r VMOVAPS ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed single precision floating-point values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.0F.W0 29 /r VMOVAPS zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1	Move aligned packed single precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

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

Description

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

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary or a general-

protection exception (#GP) will be generated. For EVEX.512 encoded versions, the operand must be aligned to the size of the memory operand. To move single precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

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

EVEX.512 encoded version:

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

VEX.256 and EVEX.256 encoded version:

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

128-bit versions:

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

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

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

Operation

VMOVAPS (EVEX Encoded Versions, Register-Copy Form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] := 0         ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVAPS (EVEX Encoded Versions, Store Form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            SRC[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*          ; merging-masking
    FI;
ENDFOR;
```

VMOVAPS (EVEX Encoded Versions, Load Form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] := 0             ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVAPS (VEX.256 Encoded Version, Load - and Register Copy)

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

VMOVAPS (VEX.256 Encoded Version, Store-Form)

```
DEST[255:0] := SRC[255:0]
```

VMOVAPS (VEX.128 Encoded Version, Load - and Register Copy)

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

MOVAPS (128-bit Load- and Register-Copy- Form Legacy SSE Version)

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

(V)MOVAPS (128-bit Store-Form Version)

```
DEST[127:0] := SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE in Table 2-18, "Type 1 Class Exception Conditions," additionally:

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Table 2-46, "Type E1 Class Exception Conditions."

MOVBE—Move Data After Swapping Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NFx 0F 38 F0 /r MOVBE r16, m16	RM	V/V	MOVBE	Reverse byte order in m16 and move to r16.
NFx 0F 38 F0 /r MOVBE r32, m32	RM	V/V	MOVBE	Reverse byte order in m32 and move to r32.
NFx REX.W + 0F 38 F0 /r MOVBE r64, m64	RM	V/N.E.	MOVBE	Reverse byte order in m64 and move to r64.
NFx 0F 38 F1 /r MOVBE m16, r16	MR	V/V	MOVBE	Reverse byte order in r16 and move to m16.
NFx 0F 38 F1 /r MOVBE m32, r32	MR	V/V	MOVBE	Reverse byte order in r32 and move to m32.
NFx REX.W + 0F 38 F1 /r MOVBE m64, r64	MR	V/N.E.	MOVBE	Reverse byte order in r64 and move to m64.

Instruction Operand Encoding

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

Description

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

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

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

Operation

```
TEMP := SRC

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

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

Flags Affected

None.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

MOVDDUP—Replicate Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 12 /r MOVDDUP xmm1, xmm2/m64	A	V/V	SSE3	Move double precision floating-point value from xmm2/m64 and duplicate into xmm1.
VEX.128.F2.0F.WIG 12 /r VMOVDDUP xmm1, xmm2/m64	A	V/V	AVX	Move double precision floating-point value from xmm2/m64 and duplicate into xmm1.
VEX.256.F2.0F.WIG 12 /r VMOVDDUP ymm1, ymm2/m256	A	V/V	AVX	Move even index double precision floating-point values from ymm2/mem and duplicate each element into ymm1.
EVEX.128.F2.0F.W1 12 /r VMOVDDUP xmm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move double precision floating-point value from xmm2/m64 and duplicate each element into xmm1 subject to writemask k1.
EVEX.256.F2.0F.W1 12 /r VMOVDDUP ymm1 {k1}{z}, ymm2/m256	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move even index double precision floating-point values from ymm2/m256 and duplicate each element into ymm1 subject to writemask k1.
EVEX.512.F2.0F.W1 12 /r VMOVDDUP zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512F OR AVX10.1	Move even index double precision floating-point values from zmm2/m512 and duplicate each element into zmm1 subject to writemask k1.

Instruction Operand Encoding

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

Description

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

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

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

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

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

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

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

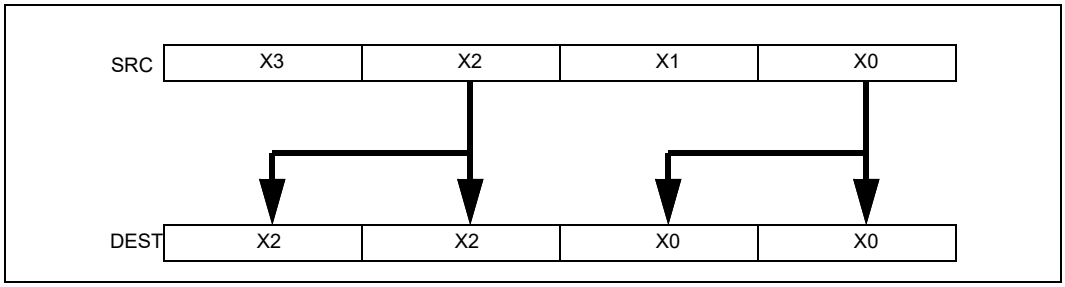


Figure 4-2. VMOVDDUP Operation

Operation

VMOVDDUP (EVEX Encoded Versions)

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

VMOVDDUP (VEX.256 Encoded Version)

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

VMOVDDUP (VEX.128 Encoded Version)

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

MOVDDUP (128-bit Legacy SSE Version)
DEST[63:0] := SRC[63:0]
DEST[127:64] := SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, "Type 5 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-54, "Type E5NF Class Exception Conditions."
Additionally:

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

MOVDIR64B—Move 64 Bytes as Direct Store

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

Instruction Operand Encoding[1]

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

Description

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

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

Unlike stores with non-temporal hint which allow UC/WP memory-type for destination to override the non-temporal hint, direct-stores always follow WC memory type protocol irrespective of destination address memory type (including UC/WP types). Unlike WC stores and stores with non-temporal hint, direct-stores are eligible for immediate eviction from the write-combining buffer, and thus not combined with younger stores (including direct-stores) to the same address. Older WC and non-temporal stores held in the write-combing buffer may be combined with younger direct stores to the same address. Direct stores are weakly ordered relative to other stores. Software that desires stronger ordering should use a fencing instruction (MFENCE or SFENCE) before or after a direct store to enforce the ordering desired.

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

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

Operation

```
DEST := SRC;
```

Intel C/C++ Compiler Intrinsic Equivalent

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

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

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If address in destination (register) operand is not aligned to a 64-byte boundary.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#UD	If CPUID.07H.00H:ECX.MOVDIR64B[28] = 0.
	If LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
	If address in destination (register) operand is not aligned to a 64-byte boundary.
#UD	If CPUID.07H.00H:ECX.MOVDIR64B[28] = 0.
	If LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If memory address referencing the SS segment is in non-canonical form.
#GP(0)	If the memory address is in non-canonical form.
	If address in destination (register) operand is not aligned to a 64-byte boundary.
#PF (fault-code)	For a page fault.
#UD	If CPUID.07H.00H:ECX.MOVDIR64B[28] = 0.
	If LOCK prefix is used.

MOVDIRI—Move Doubleword as Direct Store

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

Instruction Operand Encoding[1]

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

Description

Moves the doubleword integer in the source operand (second operand) to the destination operand (first operand) using a direct-store operation. The source operand is a general purpose register. The destination operand is a 32-bit memory location. In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See summary chart at the beginning of this section for encoding data and limits.

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

Unlike WC stores and stores with non-temporal hint, direct-stores are eligible for immediate eviction from the write-combining buffer, and thus not combined with younger stores (including direct-stores) to the same address. Older WC and non-temporal stores held in the write-combing buffer may be combined with younger direct stores to the same address. Direct stores are weakly ordered relative to other stores. Software that desires stronger ordering should use a fencing instruction (MFENCE or SFENCE) before or after a direct store to enforce the ordering desired.

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

Operation

```
DEST := SRC;
```

Intel C/C++ Compiler Intrinsic Equivalent

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

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

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#UD	If CPUID.07H.00H:ECX.MOVDIRI[27] = 0.
	If LOCK prefix or operand-size (66H) prefix is used.
#AC	If alignment checking is enabled and an unaligned memory reference made while in current privilege level 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD	If CPUID.07H.00H:ECX.MOVDIRI[27] = 0.
	If LOCK prefix or operand-size (66H) prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If memory address referencing the SS segment is in non-canonical form.
#GP(0)	If the memory address is in non-canonical form.
#PF (fault-code)	For a page fault.
#UD	If CPUID.07H.00H:ECX.MOVDIRI[27] = 0.
	If LOCK prefix or operand-size (66H) prefix is used.
#AC	If alignment checking is enabled and an unaligned memory reference made while in current privilege level 3.

MOVD/MOVQ—Move Doubleword/Move Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
NP 0F 6E /r MOVD mm, r/m32	A	V/V	MMX	Move doubleword from r/m32 to mm.
NP REX.W + 0F 6E /r MOVQ mm, r/m64	A	V/N.E.	MMX	Move quadword from r/m64 to mm.
NP 0F 7E /r MOVD r/m32, mm	B	V/V	MMX	Move doubleword from mm to r/m32.
NP REX.W + 0F 7E /r MOVQ r/m64, mm	B	V/N.E.	MMX	Move quadword from mm to r/m64.
66 0F 6E /r MOVD xmm, r/m32	A	V/V	SSE2	Move doubleword from r/m32 to xmm.
66 REX.W 0F 6E /r MOVQ xmm, r/m64	A	V/N.E.	SSE2	Move quadword from r/m64 to xmm.
66 0F 7E /r MOVD r/m32, xmm	B	V/V	SSE2	Move doubleword from xmm register to r/m32.
66 REX.W 0F 7E /r MOVQ r/m64, xmm	B	V/N.E.	SSE2	Move quadword from xmm register to r/m64.
VEX.128.66.0F.W0 6E / VMOVD xmm1, r32/m32	A	V/V	AVX	Move doubleword from r/m32 to xmm1.
VEX.128.66.0F.W1 6E /r VMOVQ xmm1, r64/m64	A	V/N.E.[1]	AVX	Move quadword from r/m64 to xmm1.
VEX.128.66.0F.W0 7E /r VMOVD r32/m32, xmm1	B	V/V	AVX	Move doubleword from xmm1 register to r/m32.
VEX.128.66.0F.W1 7E /r VMOVQ r64/m64, xmm1	B	V/N.E.[1]	AVX	Move quadword from xmm1 register to r/m64.
EVEX.128.66.0F.W0 6E /r VMOVD xmm1, r32/m32	C	V/V	AVX512F OR AVX10.1	Move doubleword from r/m32 to xmm1.
EVEX.128.66.0F.W1 6E /r VMOVQ xmm1, r64/m64	C	V/N.E.	AVX512F OR AVX10.1	Move quadword from r/m64 to xmm1.
EVEX.128.66.0F.W0 7E /r VMOVD r32/m32, xmm1	D	V/V	AVX512F OR AVX10.1	Move doubleword from xmm1 register to r/m32.
EVEX.128.66.0F.W1 7E /r VMOVQ r64/m64, xmm1	D	V/N.E.[1]	AVX512F OR AVX10.1	Move quadword from xmm1 register to r/m64.

NOTES:

1. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

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

Description

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

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

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

MOVD/Q with XMM destination:

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

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

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

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

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

Stores the low dword/qword of the source XMM register to 32/64-bit memory location or general-purpose register. Qword operation requires the use of REX.W=1, VEX.W=1, or EVEX.W=1.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

If VMOVD or VMOVQ is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

MOVD (When Destination Operand is an MMX Technology Register)

```
DEST[31:0] := SRC;
DEST[63:32] := 00000000H;
```

MOVD (When Destination Operand is an XMM Register)

```
DEST[31:0] := SRC;
DEST[127:32] := 000000000000000000000000H;
DEST[MAXVL-1:128] (Unmodified)
```

MOVD (When Source Operand is an MMX Technology or XMM Register)

```
DEST := SRC[31:0];
```

VMOVD (VEX-Encoded Version when Destination is an XMM Register)

```
DEST[31:0] := SRC[31:0]
DEST[MAXVL-1:32] := 0
```

MOVQ (When Destination Operand is an XMM Register)

```
DEST[63:0] := SRC[63:0];
DEST[127:64] := 0000000000000000H;
DEST[MAXVL-1:128] (Unmodified)
```

MOVQ (When Destination Operand is r/m64)

```
DEST[63:0] := SRC[63:0];
```

MOVQ (When Source Operand is an XMM Register or r/m64)

```
DEST := SRC[63:0];
```

VMOVQ (VEX-Encoded Version When Destination is an XMM Register)

```
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] := 0
```

VMOVD (EVEX-Encoded Version When Destination is an XMM Register)

```
DEST[31:0] := SRC[31:0]
DEST[MAXVL-1:32] := 0
```

VMOVQ (EVEX-Encoded Version When Destination is an XMM Register)

```
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVD __m64 _mm_cvtsi32_si64 (int i)
MOVD int _mm_cvtsi64_si32 (__m64m)
MOVD __m128i _mm_cvtsi32_si128 (int a)
MOVD int _mm_cvtsi128_si32 (__m128i a)
MOVQ __int64 _mm_cvtsi128_si64(__m128i);
MOVQ __m128i _mm_cvtsi64_si128(__int64);
VMOVD __m128i _mm_cvtsi32_si128(int);
VMOVD int _mm_cvtsi128_si32(__m128i);
VMOVQ __m128i _mm_cvtsi64_si128 (__int64);
VMOVQ __int64 _mm_cvtsi128_si64(__m128i);
VMOVQ __m128i _mm_loadl_epi64(__m128i * s);
VMOVQ void _mm_storel_epi64(__m128i * d, __m128i s);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-59, “Type E9NF Class Exception Conditions.”
Additionally:

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

MOVDQ2Q—Move Quadword from XMM to MMX Technology Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F2 0F D6 /r	MOVDQ2Q mm, xmm	RM	Valid	Valid	Move low quadword from xmm to mmx register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Moves the low quadword from the source operand (second operand) to the destination operand (first operand). The source operand is an XMM register and the destination operand is an MMX technology register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the MOVDQ2Q instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Operation

```
DEST := SRC[63:0];
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVDQ2Q __m64 _mm_movepi64_pi64 (__m128i a)

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.
	If CR4.OSFXSR[bit 9] = 0.
	If CPUID.01H:EDX.SSE2[26] = 0.
	If the LOCK prefix is used.
#MF	If there is a pending x87 FPU exception.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

MOVDQA,VMOVDQA32/64—Move Aligned Packed Integer Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 6F /r MOVDQA xmm1, xmm2/m128	A	V/V	SSE2	Move aligned packed integer values from xmm2/mem to xmm1.
66 0F 7F /r MOVDQA xmm2/m128, xmm1	B	V/V	SSE2	Move aligned packed integer values from xmm1 to xmm2/mem.
VEX.128.66.0F.WIG 6F /r VMOVDQA xmm1, xmm2/m128	A	V/V	AVX	Move aligned packed integer values from xmm2/mem to xmm1.
VEX.128.66.0F.WIG 7F /r VMOVDQA xmm2/m128, xmm1	B	V/V	AVX	Move aligned packed integer values from xmm1 to xmm2/mem.
VEX.256.66.0F.WIG 6F /r VMOVDQA ymm1, ymm2/m256	A	V/V	AVX	Move aligned packed integer values from ymm2/mem to ymm1.
VEX.256.66.0F.WIG 7F /r VMOVDQA ymm2/m256, ymm1	B	V/V	AVX	Move aligned packed integer values from ymm1 to ymm2/mem.
EVEX.128.66.0F.W0 6F /r VMOVDQA32 xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed doubleword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F.W0 6F /r VMOVDQA32 ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed doubleword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F.W0 6F /r VMOVDQA32 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F OR AVX10.1	Move aligned packed doubleword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.66.0F.W0 7F /r VMOVDQA32 xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed doubleword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W0 7F /r VMOVDQA32 ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed doubleword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W0 7F /r VMOVDQA32 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1	Move aligned packed doubleword integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.66.0F.W1 6F /r VMOVDQA64 xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed quadword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F.W1 6F /r VMOVDQA64 ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed quadword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F.W1 6F /r VMOVDQA64 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F OR AVX10.1	Move aligned packed quadword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.66.0F.W1 7F /r VMOVDQA64 xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed quadword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W1 7F /r VMOVDQA64 ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed quadword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W1 7F /r VMOVDQA64 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1	Move aligned packed quadword integer values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
C	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
D	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

EVEX encoded versions:

Moves 128, 256 or 512 bits of packed doubleword/quadword integer values from the source operand (the second operand) to the destination operand (the first operand). This instruction can be used to load a vector register from an int32/int64 memory location, to store the contents of a vector register into an int32/int64 memory location, or to move data between two ZMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16 (EVEX.128)/32(EVEX.256)/64(EVEX.512)-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction.

The destination operand is updated at 32-bit (VMOVDQA32) or 64-bit (VMOVDQA64) granularity according to the writemask.

VEX.256 encoded version:

Moves 256 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction. Bits (MAXVL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed.

Operation

VMOVDQA32 (EVEX Encoded Versions, Register-Copy Form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] := 0         ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVDQA32 (EVEX Encoded Versions, Store-Form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*       ; merging-masking
    FI;
ENDFOR;
```

VMOVDQA32 (EVEX Encoded Versions, Load-Form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] := 0         ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVDQA64 (EVEX Encoded Versions, Register-Copy Form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] := 0         ; zeroing-masking
            FI
    FI;
ENDFOR
```

```
DEST[MAXVL-1:VL] := 0
```

VMOVDQA64 (EVEX Encoded Versions, Store-Form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[i+63:i]
        ELSE *DEST[i+63:i] remains unchanged*          ; merging-masking
    FI;
ENDFOR;
```

VMOVDQA64 (EVEX Encoded Versions, Load-Form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] := 0             ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVDQA (VEX.256 Encoded Version, Load - and Register Copy)

```
DEST[255:0] := SRC[255:0]
DEST[MAXVL-1:256] := 0
```

VMOVDQA (VEX.256 Encoded Version, Store-Form)

```
DEST[255:0] := SRC[255:0]
```

VMOVDQA (VEX.128 Encoded Version)

```
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] := 0
```

VMOVDQA (128-bit Load- and Register-Copy- Form Legacy SSE Version)

```
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVDQA (128-bit Store-Form Version)

```
DEST[127:0] := SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVDQA32 __m512i _mm512_load_epi32(void * sa);
VMOVDQA32 __m512i _mm512_mask_load_epi32(__m512i s, __mmask16 k, void * sa);
VMOVDQA32 __m512i _mm512_maskz_load_epi32(__mmask16 k, void * sa);
VMOVDQA32 void _mm512_store_epi32(void * d, __m512i a);
VMOVDQA32 void _mm512_mask_store_epi32(void * d, __mmask16 k, __m512i a);
VMOVDQA32 __m256i _mm256_mask_load_epi32(__m256i s, __mmask8 k, void * sa);
VMOVDQA32 __m256i _mm256_maskz_load_epi32(__mmask8 k, void * sa);
VMOVDQA32 void _mm256_store_epi32(void * d, __m256i a);
VMOVDQA32 void _mm256_mask_store_epi32(void * d, __mmask8 k, __m256i a);
VMOVDQA32 __m128i _mm_mask_load_epi32(__m128i s, __mmask8 k, void * sa);
VMOVDQA32 __m128i _mm_maskz_load_epi32(__mmask8 k, void * sa);
VMOVDQA32 void _mm_store_epi32(void * d, __m128i a);
VMOVDQA32 void _mm_mask_store_epi32(void * d, __mmask8 k, __m128i a);
VMOVDQA64 __m512i _mm512_load_epi64(void * sa);
VMOVDQA64 __m512i _mm512_mask_load_epi64(__m512i s, __mmask8 k, void * sa);
VMOVDQA64 __m512i _mm512_maskz_load_epi64(__mmask8 k, void * sa);
VMOVDQA64 void _mm512_store_epi64(void * d, __m512i a);
VMOVDQA64 void _mm512_mask_store_epi64(void * d, __mmask8 k, __m512i a);
VMOVDQA64 __m256i _mm256_mask_load_epi64(__m256i s, __mmask8 k, void * sa);
VMOVDQA64 __m256i _mm256_maskz_load_epi64(__mmask8 k, void * sa);
VMOVDQA64 void _mm256_store_epi64(void * d, __m256i a);
VMOVDQA64 void _mm256_mask_store_epi64(void * d, __mmask8 k, __m256i a);
VMOVDQA64 __m128i _mm_mask_load_epi64(__m128i s, __mmask8 k, void * sa);
VMOVDQA64 __m128i _mm_maskz_load_epi64(__mmask8 k, void * sa);
VMOVDQA64 void _mm_store_epi64(void * d, __m128i a);
VMOVDQA64 void _mm_mask_store_epi64(void * d, __mmask8 k, __m128i a);
MOVDQA void __m256i _mm256_load_si256 (__m256i * p);
MOVDQA _mm256_store_si256(_m256i *p, __m256i a);
MOVDQA __m128i _mm_load_si128 (__m128i * p);
MOVDQA void _mm_store_si128(__m128i *p, __m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2 in Table 2-18, “Type 1 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-46, “Type E1 Class Exception Conditions.”
Additionally:

#UD	If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

MOVDQU,VMOVDQU8/16/32/64—Move Unaligned Packed Integer Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 6F /r MOVDQU xmm1, xmm2/m128	A	V/V	SSE2	Move unaligned packed integer values from xmm2/m128 to xmm1.
F3 0F 7F /r MOVDQU xmm2/m128, xmm1	B	V/V	SSE2	Move unaligned packed integer values from xmm1 to xmm2/m128.
VEX.128.F3.0F.WIG 6F /r VMOVDQU xmm1, xmm2/m128	A	V/V	AVX	Move unaligned packed integer values from xmm2/m128 to xmm1.
VEX.128.F3.0F.WIG 7F /r VMOVDQU xmm2/m128, xmm1	B	V/V	AVX	Move unaligned packed integer values from xmm1 to xmm2/m128.
VEX.256.F3.0F.WIG 6F /r VMOVDQU ymm1, ymm2/m256	A	V/V	AVX	Move unaligned packed integer values from ymm2/m256 to ymm1.
VEX.256.F3.0F.WIG 7F /r VMOVDQU ymm2/m256, ymm1	B	V/V	AVX	Move unaligned packed integer values from ymm1 to ymm2/m256.
EVEX.128.F2.0F.W0 6F /r VMOVDQU8 xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Move unaligned packed byte integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F2.0F.W0 6F /r VMOVDQU8 ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Move unaligned packed byte integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F2.0F.W0 6F /r VMOVDQU8 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512BW OR AVX10.1	Move unaligned packed byte integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F2.0F.W0 7F /r VMOVDQU8 xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Move unaligned packed byte integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F2.0F.W0 7F /r VMOVDQU8 ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Move unaligned packed byte integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F2.0F.W0 7F /r VMOVDQU8 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512BW OR AVX10.1	Move unaligned packed byte integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.F2.0F.W1 6F /r VMOVDQU16 xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Move unaligned packed word integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F2.0F.W1 6F /r VMOVDQU16 ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Move unaligned packed word integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F2.0F.W1 6F /r VMOVDQU16 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512BW OR AVX10.1	Move unaligned packed word integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F2.0F.W1 7F /r VMOVDQU16 xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Move unaligned packed word integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F2.0F.W1 7F /r VMOVDQU16 ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Move unaligned packed word integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F2.0F.W1 7F /r VMOVDQU16 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512BW OR AVX10.1	Move unaligned packed word integer values from zmm1 to zmm2/m512 using writemask k1.

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F.W0 6F /r VMOVDQU32 xmm1 {k1}{z}, xmm2/mm128	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed doubleword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F3.0F.W0 6F /r VMOVDQU32 ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed doubleword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F3.0F.W0 6F /r VMOVDQU32 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F OR AVX10.1	Move unaligned packed doubleword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F3.0F.W0 7F /r VMOVDQU32 xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed doubleword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F3.0F.W0 7F /r VMOVDQU32 ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed doubleword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F3.0F.W0 7F /r VMOVDQU32 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1	Move unaligned packed doubleword integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.F3.0F.W1 6F /r VMOVDQU64 xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed quadword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F3.0F.W1 6F /r VMOVDQU64 ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed quadword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F3.0F.W1 6F /r VMOVDQU64 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F OR AVX10.1	Move unaligned packed quadword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F3.0F.W1 7F /r VMOVDQU64 xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed quadword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F3.0F.W1 7F /r VMOVDQU64 ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed quadword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F3.0F.W1 7F /r VMOVDQU64 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1	Move unaligned packed quadword integer values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
C	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
D	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

EVEX encoded versions:

Moves 128, 256 or 512 bits of packed byte/word/doubleword/quadword integer values from the source operand (the second operand) to the destination operand (first operand). This instruction can be used to load a vector register from a memory location, to store the contents of a vector register into a memory location, or to move data between two vector registers.

The destination operand is updated at 8-bit (VMOVDQU8), 16-bit (VMOVDQU16), 32-bit (VMOVDQU32), or 64-bit (VMOVDQU64) granularity according to the writemask.

VEX.256 encoded version:

Moves 256 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

Bits (MAXVL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned to any alignment without causing a general-protection exception (#GP) to be generated

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed.

Operation

VMOVDQU8 (EVEX Encoded Versions, Register-Copy Form)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SRC[i+7:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE  DEST[i+7:i] := 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVDQU8 (EVEX Encoded Versions, Store-Form)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] :=
            SRC[i+7:i]
        ELSE *DEST[i+7:i] remains unchanged*          ; merging-masking
    FI;
ENDFOR;
```

VMOVDQU8 (EVEX Encoded Versions, Load-Form)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SRC[i+7:i]
        ELSE
            IF *merging-masking*                      ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE  DEST[i+7:i] := 0                ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVDQU16 (EVEX Encoded Versions, Register-Copy Form)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SRC[i+15:i]
        ELSE
            IF *merging-masking*                      ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE  DEST[i+15:i] := 0               ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVDQU16 (EVEX Encoded Versions, Store-Form)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] :=
            SRC[i+15:i]
        ELSE *DEST[i+15:i] remains unchanged*         ; merging-masking
    FI;
ENDFOR;
```

VMOVDQU16 (EVEX Encoded Versions, Load-Form)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SRC[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE  DEST[i+15:i] := 0             ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVDQU32 (EVEX Encoded Versions, Register-Copy Form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] := 0             ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVDQU32 (EVEX Encoded Versions, Store-Form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            SRC[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*       ; merging-masking
    FI;
ENDFOR;
```

VMOVDQU32 (EVEX Encoded Versions, Load-Form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] := 0             ; zeroing-masking
            FI
    FI;
ENDFOR
```

```
DEST[MAXVL-1:VL] := 0
```

VMOVDQU64 (EVEX Encoded Versions, Register-Copy Form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] := 0             ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVDQU64 (EVEX Encoded Versions, Store-Form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[i+63:i]
        ELSE *DEST[i+63:i] remains unchanged*          ; merging-masking

    FI;
ENDFOR;
```

VMOVDQU64 (EVEX Encoded Versions, Load-Form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] := 0             ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVDQU (VEX.256 Encoded Version, Load - and Register Copy)

```
DEST[255:0] := SRC[255:0]
DEST[MAXVL-1:256] := 0
```

VMOVDQU (VEX.256 Encoded Version, Store-Form)

```
DEST[255:0] := SRC[255:0]
```

```
VMOVDQU (VEX.128 encoded version)
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] := 0
```

VMOVDQU (128-bit Load- and Register-Copy- Form Legacy SSE Version)
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)

(V)MOVDQU (128-bit Store-Form Version)
DEST[127:0] := SRC[127:0]

Intel C/C++ Compiler Intrinsic Equivalent

VMOVDQU16 __m512i _mm512_mask_loadu_epi16(__m512i s, __mmask32 k, void * sa);
VMOVDQU16 __m512i _mm512_maskz_loadu_epi16(__mmask32 k, void * sa);
VMOVDQU16 void _mm512_mask_storeu_epi16(void * d, __mmask32 k, __m512i a);
VMOVDQU16 __m256i _mm256_mask_loadu_epi16(__m256i s, __mmask16 k, void * sa);
VMOVDQU16 __m256i _mm256_maskz_loadu_epi16(__mmask16 k, void * sa);
VMOVDQU16 void _mm256_mask_storeu_epi16(void * d, __mmask16 k, __m256i a);
VMOVDQU16 __m128i _mm_mask_loadu_epi16(__m128i s, __mmask8 k, void * sa);
VMOVDQU16 __m128i _mm_maskz_loadu_epi16(__mmask8 k, void * sa);
VMOVDQU16 void _mm_mask_storeu_epi16(void * d, __mmask8 k, __m128i a);
VMOVDQU32 __m512i _mm512_loadu_epi32(void * sa);
VMOVDQU32 __m512i _mm512_mask_loadu_epi32(__m512i s, __mmask16 k, void * sa);
VMOVDQU32 __m512i _mm512_maskz_loadu_epi32(__mmask16 k, void * sa);
VMOVDQU32 void _mm512_storeu_epi32(void * d, __m512i a);
VMOVDQU32 void _mm512_mask_storeu_epi32(void * d, __mmask16 k, __m512i a);
VMOVDQU32 __m256i _mm256_mask_loadu_epi32(__m256i s, __mmask8 k, void * sa);
VMOVDQU32 __m256i _mm256_maskz_loadu_epi32(__mmask8 k, void * sa);
VMOVDQU32 void _mm256_storeu_epi32(void * d, __m256i a);
VMOVDQU32 void _mm256_mask_storeu_epi32(void * d, __mmask8 k, __m256i a);
VMOVDQU32 __m128i _mm_mask_loadu_epi32(__m128i s, __mmask8 k, void * sa);
VMOVDQU32 __m128i _mm_maskz_loadu_epi32(__mmask8 k, void * sa);
VMOVDQU32 void _mm_storeu_epi32(void * d, __m128i a);
VMOVDQU32 void _mm_mask_storeu_epi32(void * d, __mmask8 k, __m128i a);
VMOVDQU64 __m512i _mm512_loadu_epi64(void * sa);
VMOVDQU64 __m512i _mm512_mask_loadu_epi64(__m512i s, __mmask8 k, void * sa);
VMOVDQU64 __m512i _mm512_maskz_loadu_epi64(__mmask8 k, void * sa);
VMOVDQU64 void _mm512_storeu_epi64(void * d, __m512i a);
VMOVDQU64 void _mm512_mask_storeu_epi64(void * d, __mmask8 k, __m512i a);
VMOVDQU64 __m256i _mm256_mask_loadu_epi64(__m256i s, __mmask8 k, void * sa);
VMOVDQU64 __m256i _mm256_maskz_loadu_epi64(__mmask8 k, void * sa);
VMOVDQU64 void _mm256_storeu_epi64(void * d, __m256i a);
VMOVDQU64 void _mm256_mask_storeu_epi64(void * d, __mmask8 k, __m256i a);
VMOVDQU64 __m128i _mm_mask_loadu_epi64(__m128i s, __mmask8 k, void * sa);
VMOVDQU64 __m128i _mm_maskz_loadu_epi64(__mmask8 k, void * sa);
VMOVDQU64 void _mm_storeu_epi64(void * d, __m128i a);
VMOVDQU64 void _mm_mask_storeu_epi64(void * d, __mmask8 k, __m128i a);
VMOVDQU8 __m512i _mm512_mask_loadu_epi8(__m512i s, __mmask64 k, void * sa);
VMOVDQU8 __m512i _mm512_maskz_loadu_epi8(__mmask64 k, void * sa);
VMOVDQU8 void _mm512_mask_storeu_epi8(void * d, __mmask64 k, __m512i a);
VMOVDQU8 __m256i _mm256_mask_loadu_epi8(__m256i s, __mmask32 k, void * sa);
VMOVDQU8 __m256i _mm256_maskz_loadu_epi8(__mmask32 k, void * sa);
VMOVDQU8 void _mm256_mask_storeu_epi8(void * d, __mmask32 k, __m256i a);
VMOVDQU8 __m128i _mm_mask_loadu_epi8(__m128i s, __mmask16 k, void * sa);
VMOVDQU8 __m128i _mm_maskz_loadu_epi8(__mmask16 k, void * sa);
VMOVDQU8 void _mm_mask_storeu_epi8(void * d, __mmask16 k, __m128i a);
MOVDQU __m256i _mm256_loadu_si256 (__m256i * p);

MOVDQU _mm256_storeu_si256(_m256i *p, __m256i a);
MOVDQU __m128i _mm_loadu_si128 (__m128i * p);
MOVDQU _mm_storeu_si128(__m128i *p, __m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, "Type E4 Class Exception Conditions."
Additionally:

#UD	If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

MOVHLPS—Move Packed Single Precision Floating-Point Values High to Low

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 12 /r MOVHLPS xmm1, xmm2	RM	V/V	SSE	Move two packed single precision floating-point values from high quadword of xmm2 to low quadword of xmm1.
VEX.128.0F.WIG 12 /r VMOVHLPS xmm1, xmm2, xmm3	RVM	V/V	AVX	Merge two packed single precision floating-point values from high quadword of xmm3 and low quadword of xmm2.
EVEX.128.0F.W0 12 /r VMOVHLPS xmm1, xmm2, xmm3	RVM	V/V	AVX512F OR AVX10.1	Merge two packed single precision floating-point values from high quadword of xmm3 and low quadword of xmm2.

Instruction Operand Encoding[1]

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
RVM	ModRM:reg (w)	VEX.vvvv (r) / EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction cannot be used for memory to register moves.

128-bit two-argument form:

Moves two packed single precision floating-point values from the high quadword of the second XMM argument (second operand) to the low quadword of the first XMM register (first argument). The quadword at bits 127:64 of the destination operand is left unchanged. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

128-bit and EVEX three-argument form:

Moves two packed single precision floating-point values from the high quadword of the third XMM argument (third operand) to the low quadword of the destination (first operand). Copies the high quadword from the second XMM argument (second operand) to the high quadword of the destination (first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

If VMOVHLPS is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

Operation

MOVHLPS (128-bit Two-Argument Form)

```
DEST[63:0] := SRC[127:64]
DEST[MAXVL-1:64] (Unmodified)
```

VMOVHLPS (128-bit Three-Argument Form - VEX & EVEX)

```
DEST[63:0] := SRC2[127:64]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVHLPS __m128 _mm_movehl_ps(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

None.

1. ModRM.MOD = 011B required.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-24, “Type 7 Class Exception Conditions,” additionally:

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E7NM.128 in Table 2-57, “Type E7NM Class Exception Conditions.”

MOVHPD—Move High Packed Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 16 /r MOVHPD xmm1, m64	A	V/V	SSE2	Move double precision floating-point value from m64 to high quadword of xmm1.
VEX.128.66.0F.WIG 16 /r VMOVHPD xmm2, xmm1, m64	B	V/V	AVX	Merge double precision floating-point value from m64 and the low quadword of xmm1.
EVEX.128.66.0F.W1 16 /r VMOVHPD xmm2, xmm1, m64	D	V/V	AVX512F OR AVX10.1	Merge double precision floating-point value from m64 and the low quadword of xmm1.
66 0F 17 /r MOVHPD m64, xmm1	C	V/V	SSE2	Move double precision floating-point value from high quadword of xmm1 to m64.
VEX.128.66.0F.WIG 17 /r VMOVHPD m64, xmm1	C	V/V	AVX	Move double precision floating-point value from high quadword of xmm1 to m64.
EVEX.128.66.0F.W1 17 /r VMOVHPD m64, xmm1	E	V/V	AVX512F OR AVX10.1	Move double precision floating-point value from high quadword of xmm1 to m64.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
D	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
E	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves a double precision floating-point value from the source 64-bit memory operand and stores it in the high 64-bits of the destination XMM register. The lower 64bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:

Loads a double precision floating-point value from the source 64-bit memory operand (the third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from the first source operand (second operand) are copied to the low 64-bits of the destination. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

128-bit store:

Stores a double precision floating-point value from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPD (store) (VEX.128.66.0F 17 /r) is legal and has the same behavior as the existing 66 0F 17 store. For VMOVHPD (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVHPD is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

Operation

MOVHPD (128-bit Legacy SSE Load)

```
DEST[63:0] (Unmodified)
DEST[127:64] := SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)
```

VMOVHPD (VEX.128 & EVEX Encoded Load)

```
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[MAXVL-1:128] := 0
```

VMOVHPD (Store)

```
DEST[63:0] := SRC[127:64]
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVHPD __m128d _mm_loadh_pd (__m128d a, double *p)
MOVHPD void _mm_storeh_pd (double *p, __m128d a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, "Type 5 Class Exception Conditions," additionally:

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Table 2-59, "Type E9NF Class Exception Conditions."

MOVHPS—Move High Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 16 /r MOVHPS xmm1, m64	A	V/V	SSE	Move two packed single precision floating-point values from m64 to high quadword of xmm1.
VEX.128.0F.WIG 16 /r VMOVHPS xmm2, xmm1, m64	B	V/V	AVX	Merge two packed single precision floating-point values from m64 and the low quadword of xmm1.
EVEX.128.0F.W0 16 /r VMOVHPS xmm2, xmm1, m64	D	V/V	AVX512F OR AVX10.1	Merge two packed single precision floating-point values from m64 and the low quadword of xmm1.
NP 0F 17 /r MOVHPS m64, xmm1	C	V/V	SSE	Move two packed single precision floating-point values from high quadword of xmm1 to m64.
VEX.128.0F.WIG 17 /r VMOVHPS m64, xmm1	C	V/V	AVX	Move two packed single precision floating-point values from high quadword of xmm1 to m64.
EVEX.128.0F.W0 17 /r VMOVHPS m64, xmm1	E	V/V	AVX512F OR AVX10.1	Move two packed single precision floating-point values from high quadword of xmm1 to m64.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
D	Tuple2	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
E	Tuple2	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves two packed single precision floating-point values from the source 64-bit memory operand and stores them in the high 64-bits of the destination XMM register. The lower 64bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:

Loads two single precision floating-point values from the source 64-bit memory operand (the third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from the first source operand (the second operand) are copied to the lower 64-bits of the destination. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

128-bit store:

Stores two packed single precision floating-point values from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPS (store) (VEX.128.0F 17 /r) is legal and has the same behavior as the existing 0F 17 store. For VMOVHPS (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVHPS is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

Operation

MOVHPS (128-bit Legacy SSE Load)

```
DEST[63:0] (Unmodified)
DEST[127:64] := SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)
```

VMOVHPS (VEX.128 and EVEX Encoded Load)

```
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[MAXVL-1:128] := 0
```

VMOVHPS (Store)

```
DEST[63:0] := SRC[127:64]
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVHPS __m128 _mm_loadh_pi (__m128 a, __m64 *p)
MOVHPS void _mm_storeh_pi (__m64 *p, __m128 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, "Type 5 Class Exception Conditions," additionally:

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Table 2-59, "Type E9NF Class Exception Conditions."

MOVLHPS—Move Packed Single Precision Floating-Point Values Low to High

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 16 /r MOVLHPS xmm1, xmm2	RM	V/V	SSE	Move two packed single precision floating-point values from low quadword of xmm2 to high quadword of xmm1.
VEX.128.0F.WIG 16 /r VMOVLHPS xmm1, xmm2, xmm3	RVM	V/V	AVX	Merge two packed single precision floating-point values from low quadword of xmm3 and low quadword of xmm2.
EVEX.128.0F.W0 16 /r VMOVLHPS xmm1, xmm2, xmm3	RVM	V/V	AVX512F OR AVX10.1	Merge two packed single precision floating-point values from low quadword of xmm3 and low quadword of xmm2.

Instruction Operand Encoding[1]

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
RVM	ModRM:reg (w)	VEX.vvvv (r) / EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction cannot be used for memory to register moves.

128-bit two-argument form:

Moves two packed single precision floating-point values from the low quadword of the second XMM argument (second operand) to the high quadword of the first XMM register (first argument). The low quadword of the destination operand is left unchanged. Bits (MAXVL-1:128) of the corresponding destination register are unmodified.

128-bit three-argument forms:

Moves two packed single precision floating-point values from the low quadword of the third XMM argument (third operand) to the high quadword of the destination (first operand). Copies the low quadword from the second XMM argument (second operand) to the low quadword of the destination (first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

If VMOVLHPS is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

Operation

MOVLHPS (128-bit Two-Argument Form)

```
DEST[63:0] (Unmodified)
DEST[127:64] := SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)
```

VMOVLHPS (128-bit Three-Argument Form - VEX & EVEX)

```
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVLHPS __m128 _mm_movelh_ps(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

None.

1. ModRM.MOD = 011B required

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-24, “Type 7 Class Exception Conditions,” additionally:

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E7NM.128 in Table 2-57, “Type E7NM Class Exception Conditions.”

MOVLPD—Move Low Packed Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 12 /r MOVLPD xmm1, m64	A	V/V	SSE2	Move double precision floating-point value from m64 to low quadword of xmm1.
VEX.128.66.0F.WIG 12 /r VMOVLPD xmm2, xmm1, m64	B	V/V	AVX	Merge double precision floating-point value from m64 and the high quadword of xmm1.
EVEX.128.66.0F.W1 12 /r VMOVLPD xmm2, xmm1, m64	D	V/V	AVX512F OR AVX10.1	Merge double precision floating-point value from m64 and the high quadword of xmm1.
66 0F 13/r MOVLPD m64, xmm1	C	V/V	SSE2	Move double precision floating-point value from low quadword of xmm1 to m64.
VEX.128.66.0F.WIG 13/r VMOVLPD m64, xmm1	C	V/V	AVX	Move double precision floating-point value from low quadword of xmm1 to m64.
EVEX.128.66.0F.W1 13/r VMOVLPD m64, xmm1	E	V/V	AVX512F OR AVX10.1	Move double precision floating-point value from low quadword of xmm1 to m64.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:r/m (r)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
D	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
E	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves a double precision floating-point value from the source 64-bit memory operand and stores it in the low 64-bits of the destination XMM register. The upper 64bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:

Loads a double precision floating-point value from the source 64-bit memory operand (third operand), merges it with the upper 64-bits of the first source XMM register (second operand), and stores it in the low 128-bits of the destination XMM register (first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

128-bit store:

Stores a double precision floating-point value from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPD (store) (VEX.128.66.0F 13 /r) is legal and has the same behavior as the existing 66 0F 13 store. For VMOVLPD (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVLPD is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

Operation

MOVLPD (128-bit Legacy SSE Load)

```
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

VMOVLPD (VEX.128 & EVEX Encoded Load)
DEST[63:0] := SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

VMOVLPD (Store)
DEST[63:0] := SRC[63:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOVLPD __m128d _mm_loadl_pd (__m128d a, double *p)
MOVLPD void _mm_storel_pd (double *p, __m128d a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, "Type 5 Class Exception Conditions," additionally:

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Table 2-59, "Type E9NF Class Exception Conditions."

MOVLPS—Move Low Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 12 /r MOVLPS xmm1, m64	A	V/V	SSE	Move two packed single precision floating-point values from m64 to low quadword of xmm1.
VEX.128.0F.WIG 12 /r VMOVLPS xmm2, xmm1, m64	B	V/V	AVX	Merge two packed single precision floating-point values from m64 and the high quadword of xmm1.
EVEX.128.0F.W0 12 /r VMOVLPS xmm2, xmm1, m64	D	V/V	AVX512F OR AVX10.1	Merge two packed single precision floating-point values from m64 and the high quadword of xmm1.
0F 13/r MOVLPS m64, xmm1	C	V/V	SSE	Move two packed single precision floating-point values from low quadword of xmm1 to m64.
VEX.128.0F.WIG 13/r VMOVLPS m64, xmm1	C	V/V	AVX	Move two packed single precision floating-point values from low quadword of xmm1 to m64.
EVEX.128.0F.W0 13/r VMOVLPS m64, xmm1	E	V/V	AVX512F OR AVX10.1	Move two packed single precision floating-point values from low quadword of xmm1 to m64.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
D	Tuple2	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
E	Tuple2	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves two packed single precision floating-point values from the source 64-bit memory operand and stores them in the low 64-bits of the destination XMM register. The upper 64bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:

Loads two packed single precision floating-point values from the source 64-bit memory operand (the third operand), merges them with the upper 64-bits of the first source operand (the second operand), and stores them in the low 128-bits of the destination register (the first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

128-bit store:

Loads two packed single precision floating-point values from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPS (store) (VEX.128.0F 13 /r) is legal and has the same behavior as the existing 0F 13 store. For VMOVLPS (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVLPS is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

Operation

MOVLPS (128-bit Legacy SSE Load)

```
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

VMOVLPS (VEX.128 & EVEX Encoded Load)
DEST[63:0] := SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

VMOVLPS (Store)
DEST[63:0] := SRC[63:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOVLPS __m128 _mm_loadl_pi (__m128 a, __m64 *p)
MOVLPS void _mm_storel_pi (__m64 *p, __m128 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, "Type 5 Class Exception Conditions," additionally:

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Table 2-59, "Type E9NF Class Exception Conditions."

MOVMSKPD—Extract Packed Double Precision Floating-Point Sign Mask

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 50 /r MOVMSKPD reg, xmm	RM	V/V	SSE2	Extract 2-bit sign mask from xmm and store in reg. The upper bits of r32 or r64 are filled with zeros.
VEX.128.66.0F.WIG 50 /r VMOVMSKPD reg, xmm2	RM	V/V	AVX	Extract 2-bit sign mask from xmm2 and store in reg. The upper bits of r32 or r64 are zeroed.
VEX.256.66.0F.WIG 50 /r VMOVMSKPD reg, ymm2	RM	V/V	AVX	Extract 4-bit sign mask from ymm2 and store in reg. The upper bits of r32 or r64 are zeroed.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Extracts the sign bits from the packed double precision floating-point values in the source operand (second operand), formats them into a 2-bit mask, and stores the mask in the destination operand (first operand). The source operand is an XMM register, and the destination operand is a general-purpose register. The mask is stored in the 2 low-order bits of the destination operand. Zero-extend the upper bits of the destination.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

128-bit versions: The source operand is a YMM register. The destination operand is a general purpose register.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a general purpose register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

(V)MOVMSKPD (128-bit Versions)

```
DEST[0] := SRC[63]
DEST[1] := SRC[127]
IF DEST = r32
    THEN DEST[31:2] := 0;
    ELSE DEST[63:2] := 0;
FI
```

VMOVMSKPD (VEX.256 Encoded Version)

```
DEST[0] := SRC[63]
DEST[1] := SRC[127]
DEST[2] := SRC[191]
DEST[3] := SRC[255]
IF DEST = r32
    THEN DEST[31:4] := 0;
    ELSE DEST[63:4] := 0;
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVMSKPD int _mm_movemask_pd (__m128d a)
VMOVMSKPD _mm256_movemask_pd(__m256d a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-24, "Type 7 Class Exception Conditions," additionally:

#UD	If VEX.vvvv ≠ 1111B.

MOVMSKPS—Extract Packed Single Precision Floating-Point Sign Mask

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
NP 0F 50 /r MOVMSKPS reg, xmm	RM	V/V	SSE	Extract 4-bit sign mask from xmm and store in reg. The upper bits of r32 or r64 are filled with zeros.
VEX.128.0F.WIG 50 /r VMOVMSKPS reg, xmm2	RM	V/V	AVX	Extract 4-bit sign mask from xmm2 and store in reg. The upper bits of r32 or r64 are zeroed.
VEX.256.0F.WIG 50 /r VMOVMSKPS reg, ymm2	RM	V/V	AVX	Extract 8-bit sign mask from ymm2 and store in reg. The upper bits of r32 or r64 are zeroed.

Instruction Operand Encoding[1]

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Extracts the sign bits from the packed single precision floating-point values in the source operand (second operand), formats them into a 4- or 8-bit mask, and stores the mask in the destination operand (first operand). The source operand is an XMM or YMM register, and the destination operand is a general-purpose register. The mask is stored in the 4 or 8 low-order bits of the destination operand. The upper bits of the destination operand beyond the mask are filled with zeros.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

128-bit versions: The source operand is a YMM register. The destination operand is a general purpose register.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a general purpose register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

```
DEST[0] := SRC[31];
DEST[1] := SRC[63];
DEST[2] := SRC[95];
DEST[3] := SRC[127];

IF DEST = r32
    THEN DEST[31:4] := ZeroExtend;
    ELSE DEST[63:4] := ZeroExtend;
FI;
```

1. ModRM.MOD = 011B required

(V)MOVMSKPS (128-bit version)

```
DEST[0] := SRC[31]
DEST[1] := SRC[63]
DEST[2] := SRC[95]
DEST[3] := SRC[127]
IF DEST = r32
    THEN DEST[31:4] := 0;
    ELSE DEST[63:4] := 0;
FI
```

VMOVMSKPS (VEX.256 encoded version)

```
DEST[0] := SRC[31]
DEST[1] := SRC[63]
DEST[2] := SRC[95]
DEST[3] := SRC[127]
DEST[4] := SRC[159]
DEST[5] := SRC[191]
DEST[6] := SRC[223]
DEST[7] := SRC[255]
IF DEST = r32
    THEN DEST[31:8] := 0;
    ELSE DEST[63:8] := 0;
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

int _mm_movemask_ps(__m128 a)
int _mm256_movemask_ps(__m256 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-24, "Type 7 Class Exception Conditions," additionally:

#UD	If VEX.vvvv ≠ 1111B.

MOVNTDQ—Store Packed Integers Using Non-Temporal Hint

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F E7 /r MOVNTDQ m128, xmm1	A	V/V	SSE2	Move packed integer values in xmm1 to m128 using non-temporal hint.
VEX.128.66.0F.WIG E7 /r VMOVNTDQ m128, xmm1	A	V/V	AVX	Move packed integer values in xmm1 to m128 using non-temporal hint.
VEX.256.66.0F.WIG E7 /r VMOVNTDQ m256, ymm1	A	V/V	AVX	Move packed integer values in ymm1 to m256 using non-temporal hint.
EVEX.128.66.0F.W0 E7 /r VMOVNTDQ m128, xmm1	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move packed integer values in xmm1 to m128 using non-temporal hint.
EVEX.256.66.0F.W0 E7 /r VMOVNTDQ m256, ymm1	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move packed integer values in zmm1 to m256 using non-temporal hint.
EVEX.512.66.0F.W0 E7 /r VMOVNTDQ m512, zmm1	B	V/V	AVX512F OR AVX10.1	Move packed integer values in zmm1 to m512 using non-temporal hint.

Instruction Operand Encoding[1]

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
B	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Moves the packed integers in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain integer data (packed bytes, words, double-words, or quadwords). The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (512-bit version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the IA-32 Intel Architecture Software Developer's Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with VMOVNTDQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

VMOVNTDQ(EVEX Encoded Versions)

```
VL = 128, 256, 512
DEST[VL-1:0] := SRC[VL-1:0]
```

1. ModRM.MOD != 011B

DEST[MAXVL-1:VL] := 0

MOVNTDQ (Legacy and VEX Versions)
DEST := SRC

Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTDQ void _mm512_stream_si512(void * p, __m512i a);
VMOVNTDQ void _mm256_stream_si256 (__m256i * p, __m256i a);
MOVNTDQ void _mm_stream_si128 (__m128i * p, __m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2 in Table 2-18, "Type 1 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-47, "Type E1NF Class Exception Conditions."
Additionally:

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

MOVNTDQA—Load Double Quadword Non-Temporal Aligned Hint

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 2A /r MOVNTDQA xmm1, m128	A	V/V	SSE4_1	Move double quadword from m128 to xmm1 using non-temporal hint if WC memory type.
VEX.128.66.0F38.WIG 2A /r VMOVNTDQA xmm1, m128	A	V/V	AVX	Move double quadword from m128 to xmm1 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 ymm1 using non-temporal hint if WC memory type.
EVEX.128.66.0F38.W0 2A /r VMOVNTDQA xmm1, m128	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move 128-bit data from m128 to xmm1 using non-temporal hint if WC memory type.
EVEX.256.66.0F38.W0 2A /r VMOVNTDQA ymm1, m256	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move 256-bit data from m256 to ymm1 using non-temporal hint if WC memory type.
EVEX.512.66.0F38.W0 2A /r VMOVNTDQA zmm1, m512	B	V/V	AVX512F OR AVX10.1	Move 512-bit data from m512 to zmm1 using non-temporal hint if WC memory type.

Instruction Operand Encoding[1]

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

MOVNTDQA loads a double quadword from the source operand (second operand) to the destination operand (first operand) using a non-temporal hint if the memory source is WC (write combining) memory type. For WC memory type, the non-temporal hint may be implemented by loading a temporary internal buffer with the equivalent of an aligned cache line without filling this data to the cache. Any memory-type aliased lines in the cache will be snooped and flushed. Subsequent MOVNTDQA reads to unread portions of the WC cache line will receive data from the temporary internal buffer if data is available. The temporary internal buffer may be flushed by the processor at any time for any reason, for example:

- A load operation other than a MOVNTDQA which references memory already resident in a temporary internal buffer.
- A non-WC reference to memory already resident in a temporary internal buffer.
- Interleaving of reads and writes to a single temporary internal buffer.
- Repeated (V)MOVNTDQA loads of a particular 16-byte item in a streaming line.
- Certain micro-architectural conditions including resource shortages, detection of a mis-speculation condition, and various fault conditions.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when reading the data from memory. Using this protocol, the processor does not read the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being read can override the non-temporal hint, if the memory address specified for the non-temporal read is not a WC memory region. Information on non-temporal reads and writes can be found in "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the Intel® 64 and IA-32 Architecture Software Developer's Manual, Volume 3A.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with a MFENCE instruction should be used in conjunction with MOVNTDQA instructions if multiple processors might use different memory types for the referenced memory locations or to synchronize reads of a processor with writes by

1. ModRM.MOD != 011B

other agents in the system. A processor's implementation of the streaming load hint does not override the effective memory type, but the implementation of the hint is processor dependent. For example, a processor implementation may choose to ignore the hint and process the instruction as a normal MOVDQA for any memory type. Alternatively, another implementation may optimize cache reads generated by MOVNTDQA on WB memory type to reduce cache evictions.

The 128-bit (V)MOVNTDQA addresses must be 16-byte aligned or the instruction will cause a #GP.

The 256-bit VMOVNTDQA addresses must be 32-byte aligned or the instruction will cause a #GP.

The 512-bit VMOVNTDQA addresses must be 64-byte aligned or the instruction will cause a #GP.

Operation

MOVNTDQA (128bit- Legacy SSE Form)

```
DEST := SRC
DEST[MAXVL-1:128] (Unmodified)
```

VMOVNTDQA (VEX.128 and EVEX.128 Encoded Form)

```
DEST := SRC
DEST[MAXVL-1:128] := 0
```

VMOVNTDQA (VEX.256 and EVEX.256 Encoded Forms)

```
DEST[255:0] := SRC[255:0]
DEST[MAXVL-1:256] := 0
```

VMOVNTDQA (EVEX.512 Encoded Form)

```
DEST[511:0] := SRC[511:0]
DEST[MAXVL-1:512] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTDQA __m512i _mm512_stream_load_si512(__m512i const* p);
MOVNTDQA __m128i _mm_stream_load_si128 (const __m128i *p);
VMOVNTDQA __m256i _mm256_stream_load_si256 (__m256i const* p);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-18, "Type 1 Class Exception Conditions."

EVEX-encoded instruction, see Table 2-47, "Type E1NF Class Exception Conditions."

Additionally:

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

MOVNTI—Store Doubleword Using Non-Temporal Hint

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C3 /r MOVNTI *m32, r32*	MR	V/V	SSE2	Move doubleword from *r32* to *m32* using non-temporal hint.
NP REX.W + 0F C3 /r MOVNTI *m64, r64*	MR	V/N.E.	SSE2	Move quadword from *r64* to *m64* using non-temporal hint.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Moves the doubleword integer in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to minimize cache pollution during the write to memory. The source operand is a general-purpose register. The destination operand is a 32-bit memory location.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTI instructions if multiple processors might use different memory types to read/write the destination memory locations.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST := SRC;
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVNTI void _mm_stream_si32 (int *p, int a)
MOVNTI void _mm_stream_si64(__int64 *p, __int64 a)

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:EDX.SSE2[26] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD	If CPUID.01H:EDX.SSE2[26] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF(fault-code)	For a page fault.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:EDX.SSE2[26] = 0.
	If the LOCK prefix is used.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

MOVNTPD—Store Packed Double Precision Floating-Point Values Using Non-Temporal Hint

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2B /r MOVNTPD m128, xmm1	A	V/V	SSE2	Move packed double precision values in xmm1 to m128 using non-temporal hint.
VEX.128.66.0F.WIG 2B /r VMOVNTPD m128, xmm1	A	V/V	AVX	Move packed double precision values in xmm1 to m128 using non-temporal hint.
VEX.256.66.0F.WIG 2B /r VMOVNTPD m256, ymm1	A	V/V	AVX	Move packed double precision values in ymm1 to m256 using non-temporal hint.
EVEX.128.66.0F.W1 2B /r VMOVNTPD m128, xmm1	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move packed double precision values in xmm1 to m128 using non-temporal hint.
EVEX.256.66.0F.W1 2B /r VMOVNTPD m256, ymm1	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move packed double precision values in ymm1 to m256 using non-temporal hint.
EVEX.512.66.0F.W1 2B /r VMOVNTPD m512, zmm1	B	V/V	AVX512F OR AVX10.1	Move packed double precision values in zmm1 to m512 using non-temporal hint.

Instruction Operand Encoding[1]

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
B	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Moves the packed double precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain packed double precision, floating-pointing data. The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the IA-32 Intel Architecture Software Developer's Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPD instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

VMOVNTPD (EVEX Encoded Versions)

```
VL = 128, 256, 512
DEST[VL-1:0] := SRC[VL-1:0]
```

1. ModRM.MOD != 011B

DEST[MAXVL-1:VL] := 0

MOVNTPD (Legacy and VEX Versions)
DEST := SRC

Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTPD void _mm512_stream_pd(double * p, __m512d a);
VMOVNTPD void _mm256_stream_pd (double * p, __m256d a);
MOVNTPD void _mm_stream_pd (double * p, __m128d a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2 in Table 2-18, "Type 1 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-47, "Type E1NF Class Exception Conditions."
Additionally:

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

MOVNTPS—Store Packed Single Precision Floating-Point Values Using Non-Temporal Hint

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 2B /r MOVNTPS m128, xmm1	A	V/V	SSE	Move packed single precision values xmm1 to mem using non-temporal hint.
VEX.128.0F.WIG 2B /r VMOVNTPS m128, xmm1	A	V/V	AVX	Move packed single precision values xmm1 to mem using non-temporal hint.
VEX.256.0F.WIG 2B /r VMOVNTPS m256, ymm1	A	V/V	AVX	Move packed single precision values ymm1 to mem using non-temporal hint.
EVEX.128.0F.W0 2B /r VMOVNTPS m128, xmm1	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move packed single precision values in xmm1 to m128 using non-temporal hint.
EVEX.256.0F.W0 2B /r VMOVNTPS m256, ymm1	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move packed single precision values in ymm1 to m256 using non-temporal hint.
EVEX.512.0F.W0 2B /r VMOVNTPS m512, zmm1	B	V/V	AVX512F OR AVX10.1	Move packed single precision values in zmm1 to m512 using non-temporal hint.

Instruction Operand Encoding[1]

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
B	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Moves the packed single precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain packed single precision, floating-pointing. The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the IA-32 Intel Architecture Software Developer's Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPS instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

VMOVNTPS (EVEX Encoded Versions)

```
VL = 128, 256, 512
DEST[VL-1:0] := SRC[VL-1:0]
DEST[MAXVL-1:VL] := 0
```

1. ModRM.MOD != 011B

MOVNTPS
DEST := SRC

Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTPS void _mm512_stream_ps(float * p, __m512d a);
MOVNTPS void _mm_stream_ps (float * p, __m128d a);
VMOVNTPS void _mm256_stream_ps (float * p, __m256 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE in Table 2-18, "Type 1 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-47, "Type E1NF Class Exception Conditions."
Additionally:

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

MOVNTQ—Store of Quadword Using Non-Temporal Hint

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F E7 /r	MOVNTQ m64, mm	MR	Valid	Valid	Move quadword from mm to m64 using non-temporal hint.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Moves the quadword in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to minimize cache pollution during the write to memory. The source operand is an MMX technology register, which is assumed to contain packed integer data (packed bytes, words, or doublewords). The destination operand is a 64-bit memory location.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
DEST := SRC;
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVNTQ void _mm_stream_pi(__m64 * p, __m64 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

MOVQ—Move Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
NP 0F 6F /r MOVQ mm, mm/m64	A	V/V	MMX	Move quadword from mm/m64 to mm.
NP 0F 7F /r MOVQ mm/m64, mm	B	V/V	MMX	Move quadword from mm to mm/m64.
F3 0F 7E /r MOVQ xmm1, xmm2/m64	A	V/V	SSE2	Move quadword from xmm2/mem64 to xmm1.
VEX.128.F3.0F.WIG 7E /r VMOVQ xmm1, xmm2/m64	A	V/V	AVX	Move quadword from xmm2 to xmm1.
EVEX.128.F3.0F.W1 7E /r VMOVQ xmm1, xmm2/m64	C	V/V	AVX512F OR AVX10.1	Move quadword from xmm2/m64 to xmm1.
66 0F D6 /r MOVQ xmm2/m64, xmm1	B	V/V	SSE2	Move quadword from xmm1 to xmm2/mem64.
VEX.128.66.0F.WIG D6 /r VMOVQ xmm1/m64, xmm2	B	V/V	AVX	Move quadword from xmm2 register to xmm1/m64.
EVEX.128.66.0F.W1 D6 /r VMOVQ xmm1/m64, xmm2	D	V/V	AVX512F OR AVX10.1	Move quadword from xmm2 register to xmm1/m64.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
C	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
D	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Copies a quadword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be MMX technology registers, XMM registers, or 64-bit memory locations. This instruction can be used to move a quadword between two MMX technology registers or between an MMX technology register and a 64-bit memory location, or to move data between two XMM registers or between an XMM register and a 64-bit memory location. The instruction cannot be used to transfer data between memory locations.

When the source operand is an XMM register, the low quadword is moved; when the destination operand is an XMM register, the quadword is stored to the low quadword of the register, and the high quadword is cleared to all 0s.

In 64-bit mode and if not encoded using VEX/EVEX, use of the REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

If VMOVQ is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

MOVQ Instruction When Operating on MMX Technology Registers and Memory Locations

```
DEST := SRC;
```

MOVQ Instruction When Source and Destination Operands are XMM Registers

```
DEST[63:0] := SRC[63:0];
```

DEST[127:64] := 0000000000000000H;

MOVQ Instruction When Source Operand is XMM Register and Destination
operand is memory location:
DEST := SRC[63:0];

MOVQ Instruction When Source Operand is Memory Location and Destination
operand is XMM register:
DEST[63:0] := SRC;
DEST[127:64] := 0000000000000000H;

VMOVQ (VEX.128.F3.0F 7E) With XMM Register Source and Destination
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] := 0

VMOVQ (VEX.128.66.0F D6) With XMM Register Source and Destination
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] := 0

VMOVQ (7E - EVEX Encoded Version) With XMM Register Source and Destination
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] := 0

VMOVQ (D6 - EVEX Encoded Version) With XMM Register Source and Destination
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] := 0

VMOVQ (7E) With Memory Source
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] := 0

VMOVQ (7E - EVEX Encoded Version) With Memory Source
DEST[63:0] := SRC[63:0]
DEST[:MAXVL-1:64] := 0

VMOVQ (D6) With Memory DEST
DEST[63:0] := SRC2[63:0]

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VMOVQ __m128i _mm_loadu_si64(void * s);
VMOVQ void _mm_storeu_si64(void * d, __m128i s);
MOVQ m128i _mm_move_epi64(__m128i a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

MOVQ2DQ—Move Quadword from MMX Technology to XMM Register

Opcode / Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F D6 /r MOVQ2DQ xmm, mm	RM	V/V	SSE2	Move quadword from mmx to low quadword of xmm.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Moves the quadword from the source operand (second operand) to the low quadword of the destination operand (first operand). The source operand is an MMX technology register and the destination operand is an XMM register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the MOVQ2DQ instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Operation

```
DEST[63:0] := SRC[63:0];
DEST[127:64] := 0000000000000000H;
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVQ2DQ__128i _mm_movpi64_epi64 (__m64 a)

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.
	If CR4.OSFXSR[bit 9] = 0.
	If CPUID.01H:EDX.SSE2[26] = 0.
	If the LOCK prefix is used.
#MF	If there is a pending x87 FPU exception.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

MOVS/MOVSB/MOVSW/MOVSD/MOVSQ—Move Data From String to String

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
A4	MOVS *m8, m8*	ZO	Valid	Valid	For legacy mode, Move byte from address DS:(E)SI to ES:(E)DI. For 64-bit mode move byte from address (R|E)SI to (R|E)DI.
A5	MOVS *m16, m16*	ZO	Valid	Valid	For legacy mode, move word from address DS:(E)SI to ES:(E)DI. For 64-bit mode move word at address (R|E)SI to (R|E)DI.
A5	MOVS *m32, m32*	ZO	Valid	Valid	For legacy mode, move dword from address DS:(E)SI to ES:(E)DI. For 64-bit mode move dword from address (R|E)SI to (R|E)DI.
REX.W + A5	MOVS *m64, m64*	ZO	Valid	N.E.	Move qword from address (R|E)SI to (R|E)DI.
A4	MOVSB	ZO	Valid	Valid	For legacy mode, Move byte from address DS:(E)SI to ES:(E)DI. For 64-bit mode move byte from address (R|E)SI to (R|E)DI.
A5	MOVSW	ZO	Valid	Valid	For legacy mode, move word from address DS:(E)SI to ES:(E)DI. For 64-bit mode move word at address (R|E)SI to (R|E)DI.
A5	MOVSD	ZO	Valid	Valid	For legacy mode, move dword from address DS:(E)SI to ES:(E)DI. For 64-bit mode move dword from address (R|E)SI to (R|E)DI.
REX.W + A5	MOVSQ	ZO	Valid	N.E.	Move qword from address (R|E)SI to (R|E)DI.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Moves the byte, word, or doubleword specified with the second operand (source operand) to the location specified with the first operand (destination operand). Both the source and destination operands are located in memory. The address of the source operand is read from the DS:ESI or the DS:SI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The address of the destination operand is read from the ES:EDI or the ES:DI registers (again depending on the address-size attribute of the instruction). The DS segment may be overridden with a segment override prefix, but the ES segment cannot be overridden.

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operands" form and the "no-operands" form. The explicit-operands form (specified with the MOVS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source and destination operands should be symbols that indicate the size and location of the source value and the destination, respectively. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source and destination operand symbols must specify the correct **type** (size) of the operands (bytes, words, or doublewords), but they do not have to specify the correct **location**. The locations of the source and destination operands are always specified by the DS:(E)SI and ES:(E)DI registers, which must be loaded correctly before the move string instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the MOVS instructions. Here also DS:(E)SI and ES:(E)DI are assumed to be the source and destination operands, respectively. The size of the source and destination operands is selected with the mnemonic: MOVSB (byte move), MOVSW (word move), or MOVSD (doubleword move).

After the move operation, the (E)SI and (E)DI registers are incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI and (E)DI register are incre-

mented; if the DF flag is 1, the (E)SI and (E)DI registers are decremented.) The registers are incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

NOTE

To improve performance, more recent processors support modifications to the processor's operation during the string store operations initiated with MOVS and MOVSB. See Section 7.3.9.3 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for additional information on fast-string operation.

The MOVS, MOVSB, MOVSW, and MOVSD instructions can be preceded by the REP prefix (see "REP—Repeat String Operation (Prefix)" for a description of the REP prefix) for block moves of ECX bytes, words, or doublewords.

In 64-bit mode, the instruction's default address size is 64 bits, 32-bit address size is supported using the prefix 67H. The 64-bit addresses are specified by RSI and RDI; 32-bit address are specified by ESI and EDI. Use of the REX.W prefix promotes doubleword operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST := SRC;
Non-64-bit Mode:
IF (Byte move)
    THEN IF DF = 0
        THEN
            (E)SI := (E)SI + 1;
            (E)DI := (E)DI + 1;
        ELSE
            (E)SI := (E)SI – 1;
            (E)DI := (E)DI – 1;
        FI;
    ELSE IF (Word move)
        THEN IF DF = 0
            (E)SI := (E)SI + 2;
            (E)DI := (E)DI + 2;
            FI;
        ELSE
            (E)SI := (E)SI – 2;
            (E)DI := (E)DI – 2;
        FI;
    ELSE IF (Doubleword move)
        THEN IF DF = 0
            (E)SI := (E)SI + 4;
            (E)DI := (E)DI + 4;
            FI;
        ELSE
            (E)SI := (E)SI – 4;
            (E)DI := (E)DI – 4;
        FI;
FI;
64-bit Mode:
IF (Byte move)
    THEN IF DF = 0
        THEN
            (R|E)SI := (R|E)SI + 1;
            (R|E)DI := (R|E)DI + 1;
        ELSE
```

```
            (R|E)SI := (R|E)SI – 1;
            (R|E)DI := (R|E)DI – 1;
        FI;
    ELSE IF (Word move)
        THEN IF DF = 0
            (R|E)SI := (R|E)SI + 2;
            (R|E)DI := (R|E)DI + 2;
            FI;
        ELSE
            (R|E)SI := (R|E)SI – 2;
            (R|E)DI := (R|E)DI – 2;
        FI;
    ELSE IF (Doubleword move)
        THEN IF DF = 0
            (R|E)SI := (R|E)SI + 4;
            (R|E)DI := (R|E)DI + 4;
            FI;
        ELSE
            (R|E)SI := (R|E)SI – 4;
            (R|E)DI := (R|E)DI – 4;
        FI;
    ELSE IF (Quadword move)
        THEN IF DF = 0
            (R|E)SI := (R|E)SI + 8;
            (R|E)DI := (R|E)DI + 8;
            FI;
        ELSE
            (R|E)SI := (R|E)SI – 8;
            (R|E)DI := (R|E)DI – 8;
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

MOVSD—Move or Merge Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 10 /r MOVSD xmm1, xmm2	A	V/V	SSE2	Move scalar double precision floating-point value from xmm2 to xmm1 register.
F2 0F 10 /r MOVSD xmm1, m64	A	V/V	SSE2	Load scalar double precision floating-point value from m64 to xmm1 register.
F2 0F 11 /r MOVSD xmm1/m64, xmm2	C	V/V	SSE2	Move scalar double precision floating-point value from xmm2 register to xmm1/m64.
VEX.LIG.F2.0F.WIG 10 /r VMOVSD xmm1, xmm2, xmm3	B	V/V	AVX	Merge scalar double precision floating-point value from xmm2 and xmm3 to xmm1 register.
VEX.LIG.F2.0F.WIG 10 /r VMOVSD xmm1, m64	D	V/V	AVX	Load scalar double precision floating-point value from m64 to xmm1 register.
VEX.LIG.F2.0F.WIG 11 /r VMOVSD xmm1, xmm2, xmm3	E	V/V	AVX	Merge scalar double precision floating-point value from xmm2 and xmm3 registers to xmm1.
VEX.LIG.F2.0F.WIG 11 /r VMOVSD m64, xmm1	C	V/V	AVX	Store scalar double precision floating-point value from xmm1 register to m64.
EVEX.LLIG.F2.0F.W1 10 /r VMOVSD xmm1 {k1}{z}, xmm2, xmm3	B	V/V	AVX512F OR AVX10.1	Merge scalar double precision floating-point value from xmm2 and xmm3 registers to xmm1 under writemask k1.
EVEX.LLIG.F2.0F.W1 10 /r VMOVSD xmm1 {k1}{z}, m64	F	V/V	AVX512F OR AVX10.1	Load scalar double precision floating-point value from m64 to xmm1 register under writemask k1.
EVEX.LLIG.F2.0F.W1 11 /r VMOVSD xmm1 {k1}{z}, xmm2, xmm3	E	V/V	AVX512F OR AVX10.1	Merge scalar double precision floating-point value from xmm2 and xmm3 registers to xmm1 under writemask k1.
EVEX.LLIG.F2.0F.W1 11 /r VMOVSD m64 {k1}, xmm1	G	V/V	AVX512F OR AVX10.1	Store scalar double precision floating-point value from xmm1 register to m64 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
D	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
E	N/A	ModRM:r/m (w)	EVEX.vvvv (r)	ModRM:reg (r)	N/A
F	Tuple1 Scalar	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
G	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Moves a scalar double precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 64-bit memory locations. This instruction can be used to move a double precision floating-point value to and from the low quadword of an XMM register and a 64-bit memory location, or to move a double precision floating-point value between the low quadwords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

Legacy version: When the source and destination operands are XMM registers, bits MAXVL:64 of the destination operand remains unchanged. When the source operand is a memory location and destination operand is an XMM

registers, the quadword at bits 127:64 of the destination operand is cleared to all 0s, bits MAXVL:128 of the destination operand remains unchanged.

VEX and EVEX encoded register-register syntax: Moves a scalar double precision floating-point value from the second source operand (the third operand) to the low quadword element of the destination operand (the first operand). Bits 127:64 of the destination operand are copied from the first source operand (the second operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX and EVEX encoded memory store syntax: When the source operand is a memory location and destination operand is an XMM registers, bits MAXVL:64 of the destination operand is cleared to all 0s.

EVEX encoded versions: The low quadword of the destination is updated according to the writemask.

Note: For VMOVSD (memory store and load forms), VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instruction will #UD.

Operation

VMOVSD (EVEX.LLIG.F2.0F 10 /r: VMOVSD xmm1, m64 With Support for 32 Registers)

```
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := SRC[63:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[MAXVL-1:64] := 0
```

VMOVSD (EVEX.LLIG.F2.0F 11 /r: VMOVSD m64, xmm1 With Support for 32 Registers)

```
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := SRC[63:0]
    ELSE    *DEST[63:0] remains unchanged*      ; merging-masking
FI;
```

VMOVSD (EVEX.LLIG.F2.0F 11 /r: VMOVSD xmm1, xmm2, xmm3)

```
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := SRC2[63:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

MOVSD (128-bit Legacy SSE Version: MOVSD xmm1, xmm2)

```
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

VMOVSD (VEX.128.F2.0F 11 /r: VMOVSD xmm1, xmm2, xmm3)
DEST[63:0] := SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

VMOVSD (VEX.128.F2.0F 10 /r: VMOVSD xmm1, xmm2, xmm3)
DEST[63:0] := SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

VMOVSD (VEX.128.F2.0F 10 /r: VMOVSD xmm1, m64)
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] := 0

MOVSD/VMOVSD (128-bit Versions: MOVSD m64, xmm1 or VMOVSD m64, xmm1)
DEST[63:0] := SRC[63:0]

MOVSD (128-bit Legacy SSE Version: MOVSD xmm1, m64)
DEST[63:0] := SRC[63:0]
DEST[127:64] := 0
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMOVSD __m128d _mm_mask_load_sd(__m128d s, __mmask8 k, double * p);
VMOVSD __m128d _mm_maskz_load_sd(__mmask8 k, double * p);
VMOVSD __m128d _mm_mask_move_sd(__m128d sh, __mmask8 k, __m128d sl, __m128d a);
VMOVSD __m128d _mm_maskz_move_sd(__mmask8 k, __m128d s, __m128d a);
VMOVSD void _mm_mask_store_sd(double * p, __mmask8 k, __m128d s);
MOVSD __m128d _mm_load_sd (double *p)
MOVSD void _mm_store_sd (double *p, __m128d a)
MOVSD __m128d _mm_move_sd (__m128d a, __m128d b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, "Type 5 Class Exception Conditions," additionally:

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Table 2-60, "Type E10 Class Exception Conditions."

MOVSHDUP—Replicate Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 16 /r MOVSHDUP xmm1, xmm2/m128	A	V/V	SSE3	Move odd index single precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.128.F3.0F.WIG 16 /r VMOVSHDUP xmm1, xmm2/m128	A	V/V	AVX	Move odd index single precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.256.F3.0F.WIG 16 /r VMOVSHDUP ymm1, ymm2/m256	A	V/V	AVX	Move odd index single precision floating-point values from ymm2/mem and duplicate each element into ymm1.
EVEX.128.F3.0F.W0 16 /r VMOVSHDUP xmm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move odd index single precision floating-point values from xmm2/m128 and duplicate each element into xmm1 under writemask.
EVEX.256.F3.0F.W0 16 /r VMOVSHDUP ymm1 {k1}{z}, ymm2/m256	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move odd index single precision floating-point values from ymm2/m256 and duplicate each element into ymm1 under writemask.
EVEX.512.F3.0F.W0 16 /r VMOVSHDUP zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512F OR AVX10.1	Move odd index single precision floating-point values from zmm2/m512 and duplicate each element into zmm1 under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Duplicates odd-indexed single precision floating-point values from the source operand (the second operand) to adjacent element pair in the destination operand (the first operand). See Figure 4-3. The source operand is an XMM, YMM or ZMM register or 128, 256 or 512-bit memory location and the destination operand is an XMM, YMM or ZMM register.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed.

VEX.256 encoded version: Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX encoded version: The destination operand is updated at 32-bit granularity according to the writemask.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

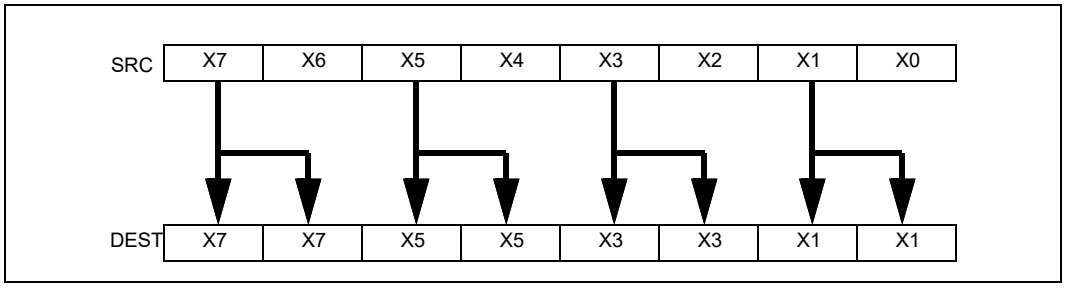


Figure 4-3. MOVSHDUP Operation

Operation

VMOVSHDUP (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
TMP_SRC[31:0] := SRC[63:32]
TMP_SRC[63:32] := SRC[63:32]
TMP_SRC[95:64] := SRC[127:96]
TMP_SRC[127:96] := SRC[127:96]
IF VL >= 256
    TMP_SRC[159:128] := SRC[191:160]
    TMP_SRC[191:160] := SRC[191:160]
    TMP_SRC[223:192] := SRC[255:224]
    TMP_SRC[255:224] := SRC[255:224]
FI;
IF VL >= 512
    TMP_SRC[287:256] := SRC[319:288]
    TMP_SRC[319:288] := SRC[319:288]
    TMP_SRC[351:320] := SRC[383:352]
    TMP_SRC[383:352] := SRC[383:352]
    TMP_SRC[415:384] := SRC[447:416]
    TMP_SRC[447:416] := SRC[447:416]
    TMP_SRC[479:448] := SRC[511:480]
    TMP_SRC[511:480] := SRC[511:480]
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_SRC[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVSHDUP (VEX.256 Encoded Version)
DEST[31:0] := SRC[63:32]
DEST[63:32] := SRC[63:32]
DEST[95:64] := SRC[127:96]
DEST[127:96] := SRC[127:96]
DEST[159:128] := SRC[191:160]
DEST[191:160] := SRC[191:160]
DEST[223:192] := SRC[255:224]
DEST[255:224] := SRC[255:224]
DEST[MAXVL-1:256] := 0

VMOVSHDUP (VEX.128 Encoded Version)
DEST[31:0] := SRC[63:32]
DEST[63:32] := SRC[63:32]
DEST[95:64] := SRC[127:96]
DEST[127:96] := SRC[127:96]
DEST[MAXVL-1:128] := 0
MOVSHDUP (128-bit Legacy SSE Version)
DEST[31:0] := SRC[63:32]
DEST[63:32] := SRC[63:32]
DEST[95:64] := SRC[127:96]
DEST[127:96] := SRC[127:96]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMOVSHDUP __m512 _mm512_movehdup_ps(__m512 a);
VMOVSHDUP __m512 _mm512_mask_movehdup_ps(__m512 s, __mmask16 k, __m512 a);
VMOVSHDUP __m512 _mm512_maskz_movehdup_ps(__mmask16 k, __m512 a);
VMOVSHDUP __m256 _mm256_mask_movehdup_ps(__m256 s, __mmask8 k, __m256 a);
VMOVSHDUP __m256 _mm256_maskz_movehdup_ps(__mmask8 k, __m256 a);
VMOVSHDUP __m128 _mm_mask_movehdup_ps(__m128 s, __mmask8 k, __m128 a);
VMOVSHDUP __m128 _mm_maskz_movehdup_ps(__mmask8 k, __m128 a);
VMOVSHDUP __m256 _mm256_movehdup_ps (__m256 a);
VMOVSHDUP __m128 _mm_movehdup_ps (__m128 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."
Additionally:

#UD	If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

MOVSLDUP—Replicate Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 12 /r MOVSLDUP xmm1, xmm2/m128	A	V/V	SSE3	Move even index single precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.128.F3.0F.WIG 12 /r VMOVSLDUP xmm1, xmm2/m128	A	V/V	AVX	Move even index single precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.256.F3.0F.WIG 12 /r VMOVSLDUP ymm1, ymm2/m256	A	V/V	AVX	Move even index single precision floating-point values from ymm2/mem and duplicate each element into ymm1.
EVEX.128.F3.0F.W0 12 /r VMOVSLDUP xmm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move even index single precision floating-point values from xmm2/m128 and duplicate each element into xmm1 under writemask.
EVEX.256.F3.0F.W0 12 /r VMOVSLDUP ymm1 {k1}{z}, ymm2/m256	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move even index single precision floating-point values from ymm2/m256 and duplicate each element into ymm1 under writemask.
EVEX.512.F3.0F.W0 12 /r VMOVSLDUP zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512F OR AVX10.1	Move even index single precision floating-point values from zmm2/m512 and duplicate each element into zmm1 under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Duplicates even-indexed single precision floating-point values from the source operand (the second operand). See Figure 4-4. The source operand is an XMM, YMM or ZMM register or 128, 256 or 512-bit memory location and the destination operand is an XMM, YMM or ZMM register.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed.

VEX.256 encoded version: Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX encoded version: The destination operand is updated at 32-bit granularity according to the writemask.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

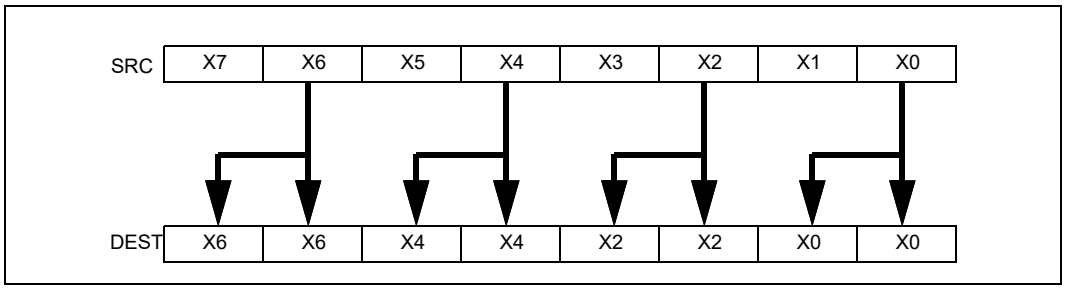


Figure 4-4. MOVSLDUP Operation

Operation

VMOVSLDUP (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
TMP_SRC[31:0] := SRC[31:0]
TMP_SRC[63:32] := SRC[31:0]
TMP_SRC[95:64] := SRC[95:64]
TMP_SRC[127:96] := SRC[95:64]
IF VL >= 256
    TMP_SRC[159:128] := SRC[159:128]
    TMP_SRC[191:160] := SRC[159:128]
    TMP_SRC[223:192] := SRC[223:192]
    TMP_SRC[255:224] := SRC[223:192]
FI;
IF VL >= 512
    TMP_SRC[287:256] := SRC[287:256]
    TMP_SRC[319:288] := SRC[287:256]
    TMP_SRC[351:320] := SRC[351:320]
    TMP_SRC[383:352] := SRC[351:320]
    TMP_SRC[415:384] := SRC[415:384]
    TMP_SRC[447:416] := SRC[415:384]
    TMP_SRC[479:448] := SRC[479:448]
    TMP_SRC[511:480] := SRC[479:448]
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_SRC[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVSLDUP (VEX.256 Encoded Version)
DEST[31:0] := SRC[31:0]
DEST[63:32] := SRC[31:0]
DEST[95:64] := SRC[95:64]
DEST[127:96] := SRC[95:64]
DEST[159:128] := SRC[159:128]
DEST[191:160] := SRC[159:128]
DEST[223:192] := SRC[223:192]
DEST[255:224] := SRC[223:192]
DEST[MAXVL-1:256] := 0

VMOVSLDUP (VEX.128 Encoded Version)
DEST[31:0] := SRC[31:0]
DEST[63:32] := SRC[31:0]
DEST[95:64] := SRC[95:64]
DEST[127:96] := SRC[95:64]
DEST[MAXVL-1:128] := 0
MOVSLDUP (128-bit Legacy SSE Version)
DEST[31:0] := SRC[31:0]
DEST[63:32] := SRC[31:0]
DEST[95:64] := SRC[95:64]
DEST[127:96] := SRC[95:64]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMOVSLDUP __m512 _mm512_moveldup_ps(__m512 a);
VMOVSLDUP __m512 _mm512_mask_moveldup_ps(__m512 s, __mmask16 k, __m512 a);
VMOVSLDUP __m512 _mm512_maskz_moveldup_ps(__mmask16 k, __m512 a);
VMOVSLDUP __m256 _mm256_mask_moveldup_ps(__m256 s, __mmask8 k, __m256 a);
VMOVSLDUP __m256 _mm256_maskz_moveldup_ps(__mmask8 k, __m256 a);
VMOVSLDUP __m128 _mm_mask_moveldup_ps(__m128 s, __mmask8 k, __m128 a);
VMOVSLDUP __m128 _mm_maskz_moveldup_ps(__mmask8 k, __m128 a);
VMOVSLDUP __m256 _mm256_moveldup_ps (__m256 a);
VMOVSLDUP __m128 _mm_moveldup_ps (__m128 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."
Additionally:

#UD	If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

MOVSS—Move or Merge Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 10 /r MOVSS xmm1, xmm2	A	V/V	SSE	Merge scalar single precision floating-point value from xmm2 to xmm1 register.
F3 0F 10 /r MOVSS xmm1, m32	A	V/V	SSE	Load scalar single precision floating-point value from m32 to xmm1 register.
VEX.LIG.F3.0F.WIG 10 /r VMOVSS xmm1, xmm2, xmm3	B	V/V	AVX	Merge scalar single precision floating-point value from xmm2 and xmm3 to xmm1 register
VEX.LIG.F3.0F.WIG 10 /r VMOVSS xmm1, m32	D	V/V	AVX	Load scalar single precision floating-point value from m32 to xmm1 register.
F3 0F 11 /r MOVSS xmm2/m32, xmm1	C	V/V	SSE	Move scalar single precision floating-point value from xmm1 register to xmm2/m32.
VEX.LIG.F3.0F.WIG 11 /r VMOVSS xmm1, xmm2, xmm3	E	V/V	AVX	Move scalar single precision floating-point value from xmm2 and xmm3 to xmm1 register.
VEX.LIG.F3.0F.WIG 11 /r VMOVSS m32, xmm1	C	V/V	AVX	Move scalar single precision floating-point value from xmm1 register to m32.
EVEX.LLIG.F3.0F.W0 10 /r VMOVSS xmm1 {k1}{z}, xmm2, xmm3	B	V/V	AVX512F OR AVX10.1	Move scalar single precision floating-point value from xmm2 and xmm3 to xmm1 register under writemask k1.
EVEX.LLIG.F3.0F.W0 10 /r VMOVSS xmm1 {k1}{z}, m32	F	V/V	AVX512F OR AVX10.1	Move scalar single precision floating-point values from m32 to xmm1 under writemask k1.
EVEX.LLIG.F3.0F.W0 11 /r VMOVSS xmm1 {k1}{z}, xmm2, xmm3	E	V/V	AVX512F OR AVX10.1	Move scalar single precision floating-point value from xmm2 and xmm3 to xmm1 register under writemask k1.
EVEX.LLIG.F3.0F.W0 11 /r VMOVSS m32 {k1}, xmm1	G	V/V	AVX512F OR AVX10.1	Move scalar single precision floating-point values from xmm1 to m32 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
D	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
E	N/A	ModRM:r/m (w)	EVEX.vvvv (r)	ModRM:reg (r)	N/A
F	Tuple1 Scalar	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
G	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Moves a scalar single precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 32-bit memory locations. This instruction can be used to move a single precision floating-point value to and from the low doubleword of an XMM register and a 32-bit memory location, or to move a single precision floating-point value between the low doublewords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

Legacy version: When the source and destination operands are XMM registers, bits (MAXVL-1:32) of the corresponding destination register are unmodified. When the source operand is a memory location and destination

operand is an XMM registers, Bits (127:32) of the destination operand is cleared to all 0s, bits MAXVL:128 of the destination operand remains unchanged.

VEX and EVEX encoded register-register syntax: Moves a scalar single precision floating-point value from the second source operand (the third operand) to the low doubleword element of the destination operand (the first operand). Bits 127:32 of the destination operand are copied from the first source operand (the second operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX and EVEX encoded memory load syntax: When the source operand is a memory location and destination operand is an XMM registers, bits MAXVL:32 of the destination operand is cleared to all 0s.

EVEX encoded versions: The low doubleword of the destination is updated according to the writemask.

Note: For memory store form instruction "VMOVSS m32, xmm1", VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD. For memory store form instruction "VMOVSS mv {k1}, xmm1", EVEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Software should ensure VMOVSS is encoded with VEX.L=0. Encoding VMOVSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VMOVSS (EVEX.LLIG.F3.0F.W0 11 /r When the Source Operand is Memory and the Destination is an XMM Register)

```
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := SRC[31:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[MAXVL-1:32] := 0
```

VMOVSS (EVEX.LLIG.F3.0F.W0 10 /r When the Source Operand is an XMM Register and the Destination is Memory)

```
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := SRC[31:0]
    ELSE    *DEST[31:0] remains unchanged*      ; merging-masking
FI;
```

VMOVSS (EVEX.LLIG.F3.0F.W0 10/11 /r Where the Source and Destination are XMM Registers)

```
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := SRC2[31:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

MOVSS (Legacy SSE Version When the Source and Destination Operands are Both XMM Registers)
DEST[31:0] := SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)

VMOVSS (VEX.128.F3.0F 11 /r Where the Destination is an XMM Register)
DEST[31:0] := SRC2[31:0]
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0

VMOVSS (VEX.128.F3.0F 10 /r Where the Source and Destination are XMM Registers)
DEST[31:0] := SRC2[31:0]
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0

VMOVSS (VEX.128.F3.0F 10 /r When the Source Operand is Memory and the Destination is an XMM Register)
DEST[31:0] := SRC[31:0]
DEST[MAXVL-1:32] := 0

MOVSS/VMOVSS (When the Source Operand is an XMM Register and the Destination is Memory)
DEST[31:0] := SRC[31:0]

MOVSS (Legacy SSE Version when the Source Operand is Memory and the Destination is an XMM Register)
DEST[31:0] := SRC[31:0]
DEST[127:32] := 0
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMOVSS __m128 _mm_mask_load_ss(__m128 s, __mmask8 k, float * p);
VMOVSS __m128 _mm_maskz_load_ss(__mmask8 k, float * p);
VMOVSS __m128 _mm_mask_move_ss(__m128 sh, __mmask8 k, __m128 sl, __m128 a);
VMOVSS __m128 _mm_maskz_move_ss(__mmask8 k, __m128 s, __m128 a);
VMOVSS void _mm_mask_store_ss(float * p, __mmask8 k, __m128 a);
MOVSS __m128 _mm_load_ss(float * p)
MOVSS void_mm_store_ss(float * p, __m128 a)
MOVSS __m128 _mm_move_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, "Type 5 Class Exception Conditions," additionally:

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Table 2-60, "Type E10 Class Exception Conditions."

MOVSX/MOVSXD—Move With Sign-Extension

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F BE /r	MOVSX r16, r/m8[1]	RM	Valid	Valid	Move byte to word with sign-extension.
0F BE /r	MOVSX r32, r/m8[1]	RM	Valid	Valid	Move byte to doubleword with sign-extension.
REX.W + 0F BE /r	MOVSX r64, r/m8[1]	RM	Valid	N.E.	Move byte to quadword with sign-extension.
0F BF /r	MOVSX r32, r/m16	RM	Valid	Valid	Move word to doubleword, with sign-extension.
REX.W + 0F BF /r	MOVSX r64, r/m16	RM	Valid	N.E.	Move word to quadword with sign-extension.
63 /r[2]	MOVSXD r16, r/m16	RM	Valid	N.E.	Move word to word with sign-extension.
63 /r[2]	MOVSXD r32, r/m32	RM	Valid	N.E.	Move doubleword to doubleword with sign-extension.
REX.W + 63 /r	MOVSXD r64, r/m32	RM	Valid	N.E.	Move doubleword to quadword with sign-extension.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.
2. The use of MOVSXD without REX.W in 64-bit mode is discouraged. Regular MOV should be used instead of using MOVSXD without REX.W.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Copies the contents of the source operand (register or memory location) to the destination operand (register) and sign extends the value to 16 or 32 bits (see Figure 7-6 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1). The size of the converted value depends on the operand-size attribute.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST := SignExtend(SRC);
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

MOVUPD—Move Unaligned Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 10 /r MOVUPD xmm1, xmm2/m128	A	V/V	SSE2	Move unaligned packed double precision floating-point from xmm2/mem to xmm1.
66 0F 11 /r MOVUPD xmm2/m128, xmm1	B	V/V	SSE2	Move unaligned packed double precision floating-point from xmm1 to xmm2/mem.
VEX.128.66.0F.WIG 10 /r VMOVUPD xmm1, xmm2/m128	A	V/V	AVX	Move unaligned packed double precision floating-point from xmm2/mem to xmm1.
VEX.128.66.0F.WIG 11 /r VMOVUPD xmm2/m128, xmm1	B	V/V	AVX	Move unaligned packed double precision floating-point from xmm1 to xmm2/mem.
VEX.256.66.0F.WIG 10 /r VMOVUPD ymm1, ymm2/m256	A	V/V	AVX	Move unaligned packed double precision floating-point from ymm2/mem to ymm1.
VEX.256.66.0F.WIG 11 /r VMOVUPD ymm2/m256, ymm1	B	V/V	AVX	Move unaligned packed double precision floating-point from ymm1 to ymm2/mem.
EVEX.128.66.0F.W1 10 /r VMOVUPD xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed double precision floating-point from xmm2/m128 to xmm1 using writemask k1.
EVEX.128.66.0F.W1 11 /r VMOVUPD xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed double precision floating-point from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W1 10 /r VMOVUPD ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed double precision floating-point from ymm2/m256 to ymm1 using writemask k1.
EVEX.256.66.0F.W1 11 /r VMOVUPD ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed double precision floating-point from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W1 10 /r VMOVUPD zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F OR AVX10.1	Move unaligned packed double precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.512.66.0F.W1 11 /r VMOVUPD zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1	Move unaligned packed double precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
C	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
D	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Note: VEX.vvvv and EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

EVEX.512 encoded version:

Moves 512 bits of packed double precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a float64 memory location, to store the contents of a ZMM register into a memory. The destination operand is updated according to the writemask.

VEX.256 encoded version:

Moves 256 bits of packed double precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. Bits (MAXVL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed double precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned on a 16-byte boundary without causing a general-protection exception (#GP) to be generated

VEX.128 and EVEX.128 encoded versions: Bits (MAXVL-1:128) of the destination register are zeroed.

Operation

VMOVUPD (EVEX Encoded Versions, Register-Copy Form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] := 0             ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVUPD (EVEX Encoded Versions, Store-Form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[i+63:i]
        ELSE *DEST[i+63:i] remains unchanged*          ; merging-masking

    FI;
ENDFOR;
```

VMOVUPD (EVEX Encoded Versions, Load-Form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] := 0             ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVUPD (VEX.256 Encoded Version, Load - and Register Copy)

```
DEST[255:0] := SRC[255:0]
DEST[MAXVL-1:256] := 0
```

VMOVUPD (VEX.256 Encoded Version, Store-Form)

```
DEST[255:0] := SRC[255:0]
```

VMOVUPD (VEX.128 Encoded Version)

```
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] := 0
```

MOVUPD (128-bit Load- and Register-Copy- Form Legacy SSE Version)

```
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVUPD (128-bit Store-Form Version)

```
DEST[127:0] := SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVUPD __m512d _mm512_loadu_pd(void * s);
VMOVUPD __m512d _mm512_mask_loadu_pd(__m512d a, __mmask8 k, void * s);
VMOVUPD __m512d _mm512_maskz_loadu_pd(__mmask8 k, void * s);
VMOVUPD void _mm512_storeu_pd(void * d, __m512d a);
VMOVUPD void _mm512_mask_storeu_pd(void * d, __mmask8 k, __m512d a);
VMOVUPD __m256d _mm256_mask_loadu_pd(__m256d s, __mmask8 k, void * m);
VMOVUPD __m256d _mm256_maskz_loadu_pd(__mmask8 k, void * m);
VMOVUPD void _mm256_mask_storeu_pd(void * d, __mmask8 k, __m256d a);
VMOVUPD __m128d _mm_mask_loadu_pd(__m128d s, __mmask8 k, void * m);
VMOVUPD __m128d _mm_maskz_loadu_pd(__mmask8 k, void * m);
VMOVUPD void _mm_mask_storeu_pd(void * d, __mmask8 k, __m128d a);
MOVUPD __m256d _mm256_loadu_pd (double * p);
MOVUPD void _mm256_storeu_pd(double *p, __m256d a);
MOVUPD __m128d _mm_loadu_pd (double * p);
MOVUPD void _mm_storeu_pd(double *p, __m128d a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
Note treatment of #AC varies; additionally:

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

MOVUPS—Move Unaligned Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 10 /r MOVUPS xmm1, xmm2/m128	A	V/V	SSE	Move unaligned packed single precision floating-point from xmm2/mem to xmm1.
NP 0F 11 /r MOVUPS xmm2/m128, xmm1	B	V/V	SSE	Move unaligned packed single precision floating-point from xmm1 to xmm2/mem.
VEX.128.0F.WIG 10 /r VMOVUPS xmm1, xmm2/m128	A	V/V	AVX	Move unaligned packed single precision floating-point from xmm2/mem to xmm1.
VEX.128.0F.WIG 11 /r VMOVUPS xmm2/m128, xmm1	B	V/V	AVX	Move unaligned packed single precision floating-point from xmm1 to xmm2/mem.
VEX.256.0F.WIG 10 /r VMOVUPS ymm1, ymm2/m256	A	V/V	AVX	Move unaligned packed single precision floating-point from ymm2/mem to ymm1.
VEX.256.0F.WIG 11 /r VMOVUPS ymm2/m256, ymm1	B	V/V	AVX	Move unaligned packed single precision floating-point from ymm1 to ymm2/mem.
EVEX.128.0F.W0 10 /r VMOVUPS xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed single precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.0F.W0 10 /r VMOVUPS ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed single precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.0F.W0 10 /r VMOVUPS zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F OR AVX10.1	Move unaligned packed single precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.0F.W0 11 /r VMOVUPS xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed single precision floating-point values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.0F.W0 11 /r VMOVUPS ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed single precision floating-point values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.0F.W0 11 /r VMOVUPS zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1	Move unaligned packed single precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
C	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
D	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Note: VEX.vvvv and EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

EVEX.512 encoded version:

Moves 512 bits of packed single precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float32 memory location, to store the contents of a ZMM register into memory. The destination operand is updated according to the writemask.

VEX.256 and EVEX.256 encoded versions:

Moves 256 bits of packed single precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. Bits (MAXVL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed single precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned without causing a general-protection exception (#GP) to be generated.

VEX.128 and EVEX.128 encoded versions: Bits (MAXVL-1:128) of the destination register are zeroed.

Operation

VMOVUPS (EVEX Encoded Versions, Register-Copy Form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] := 0             ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVUPS (EVEX Encoded Versions, Store-Form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR;
```

VMOVUPS (EVEX Encoded Versions, Load-Form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] := 0             ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVUPS (VEX.256 Encoded Version, Load - and Register Copy)

```
DEST[255:0] := SRC[255:0]
DEST[MAXVL-1:256] := 0
```

VMOVUPS (VEX.256 Encoded Version, Store-Form)

```
DEST[255:0] := SRC[255:0]
```

VMOVUPS (VEX.128 Encoded Version)

```
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] := 0
```

MOVUPS (128-bit Load- and Register-Copy- Form Legacy SSE Version)

```
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVUPS (128-bit Store-Form Version)

```
DEST[127:0] := SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVUPS __m512 _mm512_loadu_ps(void * s);
VMOVUPS __m512 _mm512_mask_loadu_ps(__m512 a, __mmask16 k, void * s);
VMOVUPS __m512 _mm512_maskz_loadu_ps(__mmask16 k, void * s);
VMOVUPS void _mm512_storeu_ps(void * d, __m512 a);
VMOVUPS void _mm512_mask_storeu_ps(void * d, __mmask8 k, __m512 a);
VMOVUPS __m256 _mm256_mask_loadu_ps(__m256 a, __mmask8 k, void * s);
VMOVUPS __m256 _mm256_maskz_loadu_ps(__mmask8 k, void * s);
VMOVUPS void _mm256_mask_storeu_ps(void * d, __mmask8 k, __m256 a);
VMOVUPS __m128 _mm_mask_loadu_ps(__m128 a, __mmask8 k, void * s);
VMOVUPS __m128 _mm_maskz_loadu_ps(__mmask8 k, void * s);
VMOVUPS void _mm_mask_storeu_ps(void * d, __mmask8 k, __m128 a);
MOVUPS __m256 _mm256_loadu_ps (float * p);
MOVUPS void _mm256 _storeu_ps(float *p, __m256 a);
MOVUPS __m128 _mm_loadu_ps (float * p);
MOVUPS void _mm_storeu_ps(float *p, __m128 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
Note treatment of #AC varies.

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

MOVZX—Move With Zero-Extend

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F B6 /r	MOVZX r16, r/m8[1]	RM	Valid	Valid	Move byte to word with zero-extension.
0F B6 /r	MOVZX r32, r/m8[1]	RM	Valid	Valid	Move byte to doubleword, zero-extension.
REX.W + 0F B6 /r	MOVZX r64, r/m8[1]	RM	Valid	N.E.	Move byte to quadword, zero-extension.
0F B7 /r	MOVZX r32, r/m16	RM	Valid	Valid	Move word to doubleword, zero-extension.
REX.W + 0F B7 /r	MOVZX r64, r/m16	RM	Valid	N.E.	Move word to quadword, zero-extension.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Copies the contents of the source operand (register or memory location) to the destination operand (register) and zero extends the value. The size of the converted value depends on the operand-size attribute.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST := ZeroExtend(SRC);
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

MPSADBW—Compute Multiple Packed Sums of Absolute Difference

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 42 /r ib MPSADBW xmm1, xmm2/m128, imm8	RMI	V/V	SSE4_1	Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in xmm1 and xmm2/m128 and writes the results in xmm1. Starting offsets within xmm1 and xmm2/m128 are determined by imm8.
VEX.128.66.0F3A.WIG 42 /r ib VMPSADBW xmm1, xmm2, xmm3/m128, imm8	RVMI	V/V	AVX	Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in xmm2 and xmm3/m128 and writes the results in xmm1. Starting offsets within xmm2 and xmm3/m128 are determined by imm8.
VEX.256.66.0F3A.WIG 42 /r ib VMPSADBW ymm1, ymm2, ymm3/m256, imm8	RVMI	V/V	AVX2	Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in xmm2 and ymm3/m128 and writes the results in ymm1. Starting offsets within ymm2 and xmm3/m128 are determined by imm8.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

(V)MPSADBW calculates packed word results of sum-absolute-difference (SAD) of unsigned bytes from two blocks of 32-bit dword elements, using two select fields in the immediate byte to select the offsets of the two blocks within the first source operand and the second operand. Packed SAD word results are calculated within each 128-bit lane. Each SAD word result is calculated between a stationary block_2 (whose offset within the second source operand is selected by a two bit select control, multiplied by 32 bits) and a sliding block_1 at consecutive byte-granular position within the first source operand. The offset of the first 32-bit block of block_1 is selectable using a one bit select control, multiplied by 32 bits.

128-bit Legacy SSE version: Imm8[1:0]*32 specifies the bit offset of block_2 within the second source operand. Imm[2]*32 specifies the initial bit offset of the block_1 within the first source operand. The first source operand and destination operand are the same. The first source and destination operands are XMM registers. The second source operand is either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. Bits 7:3 of the immediate byte are ignored.

VEX.128 encoded version: Imm8[1:0]*32 specifies the bit offset of block_2 within the second source operand. Imm[2]*32 specifies the initial bit offset of the block_1 within the first source operand. The first source and destination operands are XMM registers. The second source operand is either an XMM register or a 128-bit memory location. Bits (127:128) of the corresponding YMM register are zeroed. Bits 7:3 of the immediate byte are ignored.

VEX.256 encoded version: The sum-absolute-difference (SAD) operation is repeated 8 times for MPSADW between the same block_2 (fixed offset within the second source operand) and a variable block_1 (offset is shifted by 8 bits for each SAD operation) in the first source operand. Each 16-bit result of eight SAD operations between block_2 and block_1 is written to the respective word in the lower 128 bits of the destination operand.

Additionally, VMPSADBW performs another eight SAD operations on block_4 of the second source operand and block_3 of the first source operand. (Imm8[4:3]*32 + 128) specifies the bit offset of block_4 within the second source operand. (Imm[5]*32+128) specifies the initial bit offset of the block_3 within the first source operand. Each 16-bit result of eight SAD operations between block_4 and block_3 is written to the respective word in the upper 128 bits of the destination operand.

The first source operand is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits 7:6 of the immediate byte are ignored.

Note: If VMPSADBW is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

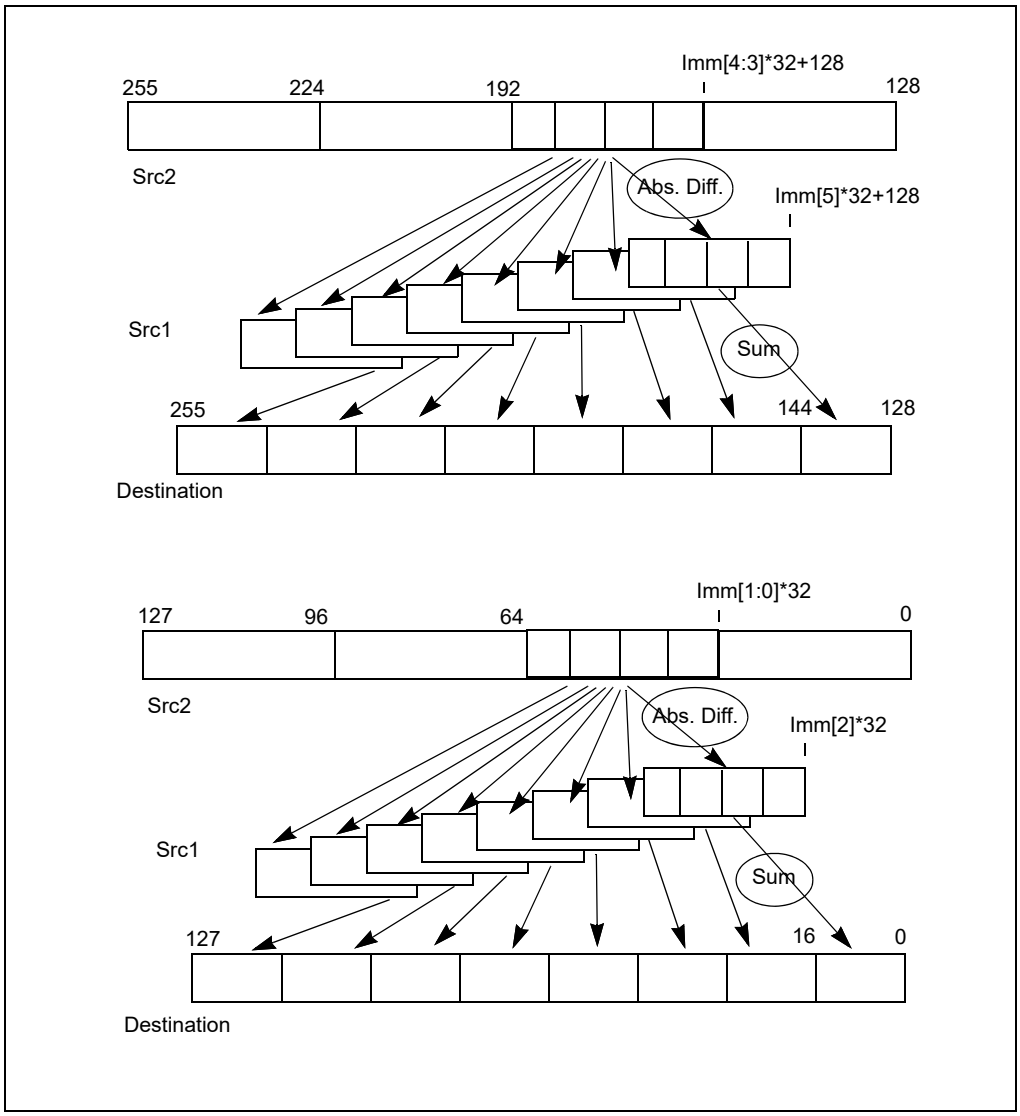


Figure 4-5. 256-bit VMPSADBW Operation

Operation

VMPSADBW (VEX.256 Encoded Version)

```
BLK2_OFFSET := imm8[1:0]*32
BLK1_OFFSET := imm8[2]*32
SRC1_BYTE0 := SRC1[BLK1_OFFSET+7:BLK1_OFFSET]
SRC1_BYTE1 := SRC1[BLK1_OFFSET+15:BLK1_OFFSET+8]
SRC1_BYTE2 := SRC1[BLK1_OFFSET+23:BLK1_OFFSET+16]
SRC1_BYTE3 := SRC1[BLK1_OFFSET+31:BLK1_OFFSET+24]
SRC1_BYTE4 := SRC1[BLK1_OFFSET+39:BLK1_OFFSET+32]
SRC1_BYTE5 := SRC1[BLK1_OFFSET+47:BLK1_OFFSET+40]
```

```
SRC1_BYTE6 := SRC1[BLK1_OFFSET+55:BLK1_OFFSET+48]
SRC1_BYTE7 := SRC1[BLK1_OFFSET+63:BLK1_OFFSET+56]
SRC1_BYTE8 := SRC1[BLK1_OFFSET+71:BLK1_OFFSET+64]
SRC1_BYTE9 := SRC1[BLK1_OFFSET+79:BLK1_OFFSET+72]
SRC1_BYTE10 := SRC1[BLK1_OFFSET+87:BLK1_OFFSET+80]
SRC2_BYTE0 := SRC2[BLK2_OFFSET+7:BLK2_OFFSET]
SRC2_BYTE1 := SRC2[BLK2_OFFSET+15:BLK2_OFFSET+8]
SRC2_BYTE2 := SRC2[BLK2_OFFSET+23:BLK2_OFFSET+16]
SRC2_BYTE3 := SRC2[BLK2_OFFSET+31:BLK2_OFFSET+24]

TEMP0 := ABS(SRC1_BYTE0 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE1 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE2 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE3 - SRC2_BYTE3)
DEST[15:0] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE1 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE2 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE3 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE4 - SRC2_BYTE3)
DEST[31:16] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE2 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE3 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE4 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE5 - SRC2_BYTE3)
DEST[47:32] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE3 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE4 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE5 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE6 - SRC2_BYTE3)
DEST[63:48] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE4 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE5 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE6 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE7 - SRC2_BYTE3)
DEST[79:64] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE5 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE6 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE7 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE8 - SRC2_BYTE3)
DEST[95:80] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE6 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE7 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE8 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE9 - SRC2_BYTE3)
DEST[111:96] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE7 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE8 - SRC2_BYTE1)
```

```
TEMP2 := ABS(SRC1_BYTE9 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE10 - SRC2_BYTE3)
DEST[127:112] := TEMP0 + TEMP1 + TEMP2 + TEMP3

BLK2_OFFSET := imm8[4:3]*32 + 128
BLK1_OFFSET := imm8[5]*32 + 128
SRC1_BYTE0 := SRC1[BLK1_OFFSET+7:BLK1_OFFSET]
SRC1_BYTE1 := SRC1[BLK1_OFFSET+15:BLK1_OFFSET+8]
SRC1_BYTE2 := SRC1[BLK1_OFFSET+23:BLK1_OFFSET+16]
SRC1_BYTE3 := SRC1[BLK1_OFFSET+31:BLK1_OFFSET+24]
SRC1_BYTE4 := SRC1[BLK1_OFFSET+39:BLK1_OFFSET+32]
SRC1_BYTE5 := SRC1[BLK1_OFFSET+47:BLK1_OFFSET+40]
SRC1_BYTE6 := SRC1[BLK1_OFFSET+55:BLK1_OFFSET+48]
SRC1_BYTE7 := SRC1[BLK1_OFFSET+63:BLK1_OFFSET+56]
SRC1_BYTE8 := SRC1[BLK1_OFFSET+71:BLK1_OFFSET+64]
SRC1_BYTE9 := SRC1[BLK1_OFFSET+79:BLK1_OFFSET+72]
SRC1_BYTE10 := SRC1[BLK1_OFFSET+87:BLK1_OFFSET+80]

SRC2_BYTE0 := SRC2[BLK2_OFFSET+7:BLK2_OFFSET]
SRC2_BYTE1 := SRC2[BLK2_OFFSET+15:BLK2_OFFSET+8]
SRC2_BYTE2 := SRC2[BLK2_OFFSET+23:BLK2_OFFSET+16]
SRC2_BYTE3 := SRC2[BLK2_OFFSET+31:BLK2_OFFSET+24]

TEMP0 := ABS(SRC1_BYTE0 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE1 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE2 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE3 - SRC2_BYTE3)
DEST[143:128] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE1 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE2 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE3 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE4 - SRC2_BYTE3)
DEST[159:144] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE2 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE3 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE4 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE5 - SRC2_BYTE3)
DEST[175:160] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE3 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE4 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE5 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE6 - SRC2_BYTE3)
DEST[191:176] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE4 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE5 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE6 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE7 - SRC2_BYTE3)
DEST[207:192] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE5 - SRC2_BYTE0)
```

```
TEMP1 := ABS(SRC1_BYTE6 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE7 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE8 - SRC2_BYTE3)
DEST[223:208] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE6 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE7 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE8 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE9 - SRC2_BYTE3)
DEST[239:224] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE7 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE8 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE9 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE10 - SRC2_BYTE3)
DEST[255:240] := TEMP0 + TEMP1 + TEMP2 + TEMP3
```

VMPSADBW (VEX.128 Encoded Version)

```
BLK2_OFFSET := imm8[1:0]*32
BLK1_OFFSET := imm8[2]*32
SRC1_BYTE0 := SRC1[BLK1_OFFSET+7:BLK1_OFFSET]
SRC1_BYTE1 := SRC1[BLK1_OFFSET+15:BLK1_OFFSET+8]
SRC1_BYTE2 := SRC1[BLK1_OFFSET+23:BLK1_OFFSET+16]
SRC1_BYTE3 := SRC1[BLK1_OFFSET+31:BLK1_OFFSET+24]
SRC1_BYTE4 := SRC1[BLK1_OFFSET+39:BLK1_OFFSET+32]
SRC1_BYTE5 := SRC1[BLK1_OFFSET+47:BLK1_OFFSET+40]
SRC1_BYTE6 := SRC1[BLK1_OFFSET+55:BLK1_OFFSET+48]
SRC1_BYTE7 := SRC1[BLK1_OFFSET+63:BLK1_OFFSET+56]
SRC1_BYTE8 := SRC1[BLK1_OFFSET+71:BLK1_OFFSET+64]
SRC1_BYTE9 := SRC1[BLK1_OFFSET+79:BLK1_OFFSET+72]
SRC1_BYTE10 := SRC1[BLK1_OFFSET+87:BLK1_OFFSET+80]

SRC2_BYTE0 := SRC2[BLK2_OFFSET+7:BLK2_OFFSET]
SRC2_BYTE1 := SRC2[BLK2_OFFSET+15:BLK2_OFFSET+8]
SRC2_BYTE2 := SRC2[BLK2_OFFSET+23:BLK2_OFFSET+16]
SRC2_BYTE3 := SRC2[BLK2_OFFSET+31:BLK2_OFFSET+24]

TEMP0 := ABS(SRC1_BYTE0 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE1 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE2 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE3 - SRC2_BYTE3)
DEST[15:0] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE1 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE2 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE3 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE4 - SRC2_BYTE3)
DEST[31:16] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE2 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE3 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE4 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE5 - SRC2_BYTE3)
DEST[47:32] := TEMP0 + TEMP1 + TEMP2 + TEMP3
```

```
TEMP0 := ABS(SRC1_BYTE3 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE4 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE5 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE6 - SRC2_BYTE3)
DEST[63:48] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE4 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE5 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE6 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE7 - SRC2_BYTE3)
DEST[79:64] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE5 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE6 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE7 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE8 - SRC2_BYTE3)
DEST[95:80] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE6 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE7 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE8 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE9 - SRC2_BYTE3)
DEST[111:96] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE7 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE8 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE9 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE10 - SRC2_BYTE3)
DEST[127:112] := TEMP0 + TEMP1 + TEMP2 + TEMP3
DEST[MAXVL-1:128] := 0
```

MPSADBW (128-bit Legacy SSE Version)

```
SRC_OFFSET := imm8[1:0]*32
DEST_OFFSET := imm8[2]*32
DEST_BYTE0 := DEST[DEST_OFFSET+7:DEST_OFFSET]
DEST_BYTE1 := DEST[DEST_OFFSET+15:DEST_OFFSET+8]
DEST_BYTE2 := DEST[DEST_OFFSET+23:DEST_OFFSET+16]
DEST_BYTE3 := DEST[DEST_OFFSET+31:DEST_OFFSET+24]
DEST_BYTE4 := DEST[DEST_OFFSET+39:DEST_OFFSET+32]
DEST_BYTE5 := DEST[DEST_OFFSET+47:DEST_OFFSET+40]
DEST_BYTE6 := DEST[DEST_OFFSET+55:DEST_OFFSET+48]
DEST_BYTE7 := DEST[DEST_OFFSET+63:DEST_OFFSET+56]
DEST_BYTE8 := DEST[DEST_OFFSET+71:DEST_OFFSET+64]
DEST_BYTE9 := DEST[DEST_OFFSET+79:DEST_OFFSET+72]
DEST_BYTE10 := DEST[DEST_OFFSET+87:DEST_OFFSET+80]

SRC_BYTE0 := SRC[SRC_OFFSET+7:SRC_OFFSET]
SRC_BYTE1 := SRC[SRC_OFFSET+15:SRC_OFFSET+8]
SRC_BYTE2 := SRC[SRC_OFFSET+23:SRC_OFFSET+16]
SRC_BYTE3 := SRC[SRC_OFFSET+31:SRC_OFFSET+24]

TEMP0 := ABS( DEST_BYTE0 - SRC_BYTE0)
TEMP1 := ABS( DEST_BYTE1 - SRC_BYTE1)
TEMP2 := ABS( DEST_BYTE2 - SRC_BYTE2)
```

```
TEMP3 := ABS( DEST_BYTE3 - SRC_BYTE3)
DEST[15:0] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS( DEST_BYTE1 - SRC_BYTE0)
TEMP1 := ABS( DEST_BYTE2 - SRC_BYTE1)
TEMP2 := ABS( DEST_BYTE3 - SRC_BYTE2)
TEMP3 := ABS( DEST_BYTE4 - SRC_BYTE3)
DEST[31:16] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS( DEST_BYTE2 - SRC_BYTE0)
TEMP1 := ABS( DEST_BYTE3 - SRC_BYTE1)
TEMP2 := ABS( DEST_BYTE4 - SRC_BYTE2)
TEMP3 := ABS( DEST_BYTE5 - SRC_BYTE3)
DEST[47:32] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS( DEST_BYTE3 - SRC_BYTE0)
TEMP1 := ABS( DEST_BYTE4 - SRC_BYTE1)
TEMP2 := ABS( DEST_BYTE5 - SRC_BYTE2)
TEMP3 := ABS( DEST_BYTE6 - SRC_BYTE3)
DEST[63:48] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS( DEST_BYTE4 - SRC_BYTE0)
TEMP1 := ABS( DEST_BYTE5 - SRC_BYTE1)
TEMP2 := ABS( DEST_BYTE6 - SRC_BYTE2)
TEMP3 := ABS( DEST_BYTE7 - SRC_BYTE3)
DEST[79:64] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS( DEST_BYTE5 - SRC_BYTE0)
TEMP1 := ABS( DEST_BYTE6 - SRC_BYTE1)
TEMP2 := ABS( DEST_BYTE7 - SRC_BYTE2)
TEMP3 := ABS( DEST_BYTE8 - SRC_BYTE3)
DEST[95:80] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS( DEST_BYTE6 - SRC_BYTE0)
TEMP1 := ABS( DEST_BYTE7 - SRC_BYTE1)
TEMP2 := ABS( DEST_BYTE8 - SRC_BYTE2)
TEMP3 := ABS( DEST_BYTE9 - SRC_BYTE3)
DEST[111:96] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS( DEST_BYTE7 - SRC_BYTE0)
TEMP1 := ABS( DEST_BYTE8 - SRC_BYTE1)
TEMP2 := ABS( DEST_BYTE9 - SRC_BYTE2)
TEMP3 := ABS( DEST_BYTE10 - SRC_BYTE3)
DEST[127:112] := TEMP0 + TEMP1 + TEMP2 + TEMP3
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)MPSADBW __m128i _mm_mpsadbw_epu8 (__m128i s1, __m128i s2, const int mask);
VMPSADBW __m256i _mm256_mpsadbw_epu8 (__m256i s1, __m256i s2, const int mask);

Flags Affected

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions."

MUL—Unsigned Multiply

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /4	MUL r/m8[1]	M	Valid	Valid	Unsigned multiply (AX := AL ∗ r/m8).
F7 /4	MUL r/m16	M	Valid	Valid	Unsigned multiply (DX:AX := AX ∗ r/m16).
F7 /4	MUL r/m32	M	Valid	Valid	Unsigned multiply (EDX:EAX := EAX ∗ r/m32).
REX.W + F7 /4	MUL r/m64	M	Valid	N.E.	Unsigned multiply (RDX:RAX := RAX ∗ r/m64).

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

Description

Performs an unsigned multiplication of the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand is an implied operand located in register AL, AX or EAX (depending on the size of the operand); the source operand is located in a general-purpose register or a memory location. The action of this instruction and the location of the result depends on the opcode and the operand size as shown in Table 4-9.

The result is stored in register AX, register pair DX:AX, or register pair EDX:EAX (depending on the operand size), with the high-order bits of the product contained in register AH, DX, or EDX, respectively. If the high-order bits of the product are 0, the CF and OF flags are cleared; otherwise, the flags are set.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

See the summary chart at the beginning of this section for encoding data and limits.

Table 4-9. MUL Results

Operand Size	Source 1	Source 2	Destination
Byte	AL	r/m8	AX
Word	AX	r/m16	DX:AX
Doubleword	EAX	r/m32	EDX:EAX
Quadword	RAX	r/m64	RDX:RAX

Operation

```
IF (Byte operation)
    THEN
        AX := AL ∗ SRC;
    ELSE (* Word or doubleword operation *)
        IF OperandSize = 16
            THEN
                DX:AX := AX ∗ SRC;
            ELSE IF OperandSize = 32
                THEN EDX:EAX := EAX ∗ SRC; FI;
            ELSE (* OperandSize = 64 *)
                RDX:RAX := RAX ∗ SRC;
        FI;
```

FI;

Flags Affected

The OF and CF flags are set to 0 if the upper half of the result is 0; otherwise, they are set to 1. The SF, ZF, AF, and PF flags are undefined.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

MULPD—Multiply Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 59 /r MULPD xmm1, xmm2/m128	A	V/V	SSE2	Multiply packed double precision floating-point values in xmm2/m128 with xmm1 and store result in xmm1.
VEX.128.66.0F.WIG 59 /r VMULPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Multiply packed double precision floating-point values in xmm3/m128 with xmm2 and store result in xmm1.
VEX.256.66.0F.WIG 59 /r VMULPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Multiply packed double precision floating-point values in ymm3/m256 with ymm2 and store result in ymm1.
EVEX.128.66.0F.W1 59 /r VMULPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1.
EVEX.256.66.0F.W1 59 /r VMULPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1.
EVEX.512.66.0F.W1 59 /r VMULPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	C	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values in zmm3/m512/m64bcst with zmm2 and store result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiply packed double precision floating-point values from the first source operand with corresponding values in the second source operand, and stores the packed double precision floating-point results in the destination operand.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the destination YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VMULPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := SRC1[i+63:i] * SRC2[63:0]
                ELSE
                    DEST[i+63:i] := SRC1[i+63:i] * SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMULPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0] * SRC2[63:0]
DEST[127:64] := SRC1[127:64] * SRC2[127:64]
DEST[191:128] := SRC1[191:128] * SRC2[191:128]
DEST[255:192] := SRC1[255:192] * SRC2[255:192]
DEST[MAXVL-1:256] := 0;
```

.

VMULPD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] * SRC2[63:0]
DEST[127:64] := SRC1[127:64] * SRC2[127:64]
DEST[MAXVL-1:128] := 0
```

MULPD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] * SRC[63:0]
DEST[127:64] := DEST[127:64] * SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VMULPD __m512d _mm512_mul_pd(__m512d a, __m512d b);
VMULPD __m512d _mm512_mask_mul_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VMULPD __m512d _mm512_maskz_mul_pd(__mmask8 k, __m512d a, __m512d b);
VMULPD __m512d _mm512_mul_round_pd(__m512d a, __m512d b, int);
VMULPD __m512d _mm512_mask_mul_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VMULPD __m512d _mm512_maskz_mul_round_pd(__mmask8 k, __m512d a, __m512d b, int);
VMULPD __m256d _mm256_mul_pd (__m256d a, __m256d b);
MULPD __m128d _mm_mul_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, "Type 2 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-48, "Type E2 Class Exception Conditions."

MULPS—Multiply Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 59 /r MULPS xmm1, xmm2/m128	A	V/V	SSE	Multiply packed single precision floating-point values in xmm2/m128 with xmm1 and store result in xmm1.
VEX.128.0F.WIG 59 /r VMULPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Multiply packed single precision floating-point values in xmm3/m128 with xmm2 and store result in xmm1.
VEX.256.0F.WIG 59 /r VMULPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Multiply packed single precision floating-point values in ymm3/m256 with ymm2 and store result in ymm1.
EVEX.128.0F.W0 59 /r VMULPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm3/m128/m32bcst to xmm2 and store result in xmm1.
EVEX.256.0F.W0 59 /r VMULPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm3/m256/m32bcst to ymm2 and store result in ymm1.
EVEX.512.0F.W0 59 /r VMULPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst {er}	C	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values in zmm3/m512/m32bcst with zmm2 and store result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiply the packed single precision floating-point values from the first source operand with the corresponding values in the second source operand, and stores the packed double precision floating-point results in the destination operand.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the destination YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VMULPS (EVEX Encoded Version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] := SRC1[i+31:i] * SRC2[31:0]
                ELSE
                    DEST[i+31:i] := SRC1[i+31:i] * SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMULPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[31:0] * SRC2[31:0]
DEST[63:32] := SRC1[63:32] * SRC2[63:32]
DEST[95:64] := SRC1[95:64] * SRC2[95:64]
DEST[127:96] := SRC1[127:96] * SRC2[127:96]
DEST[159:128] := SRC1[159:128] * SRC2[159:128]
DEST[191:160] := SRC1[191:160] * SRC2[191:160]
DEST[223:192] := SRC1[223:192] * SRC2[223:192]
DEST[255:224] := SRC1[255:224] * SRC2[255:224].
DEST[MAXVL-1:256] := 0;
```

VMULPS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] * SRC2[31:0]
DEST[63:32] := SRC1[63:32] * SRC2[63:32]
DEST[95:64] := SRC1[95:64] * SRC2[95:64]
DEST[127:96] := SRC1[127:96] * SRC2[127:96]
DEST[MAXVL-1:128] := 0
```

MULPS (128-bit Legacy SSE Version)

```
DEST[31:0] := SRC1[31:0] * SRC2[31:0]
DEST[63:32] := SRC1[63:32] * SRC2[63:32]
DEST[95:64] := SRC1[95:64] * SRC2[95:64]
DEST[127:96] := SRC1[127:96] * SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VMULPS __m512 _mm512_mul_ps(__m512 a, __m512 b);
VMULPS __m512 _mm512_mask_mul_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VMULPS __m512 _mm512_maskz_mul_ps(__mmask16 k, __m512 a, __m512 b);
VMULPS __m512 _mm512_mul_round_ps(__m512 a, __m512 b, int);
VMULPS __m512 _mm512_mask_mul_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VMULPS __m512 _mm512_maskz_mul_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VMULPS __m256 _mm256_mask_mul_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VMULPS __m256 _mm256_maskz_mul_ps(__mmask8 k, __m256 a, __m256 b);
VMULPS __m128 _mm_mask_mul_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMULPS __m128 _mm_maskz_mul_ps(__mmask8 k, __m128 a, __m128 b);
VMULPS __m256 _mm256_mul_ps (__m256 a, __m256 b);
MULPS __m128 _mm_mul_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-48, “Type E2 Class Exception Conditions.”

MULSD—Multiply Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 59 /r MULSD xmm1,xmm2/m64	A	V/V	SSE2	Multiply the low double precision floating-point value in xmm2/m64 by low double precision floating-point value in xmm1.
VEX.LIG.F2.0F.WIG 59 /r VMULSD xmm1,xmm2, xmm3/m64	B	V/V	AVX	Multiply the low double precision floating-point value in xmm3/m64 by low double precision floating-point value in xmm2.
EVEX.LLIG.F2.0F.W1 59 /r VMULSD xmm1 {k1}{z}, xmm2, xmm3/m64 {er}	C	V/V	AVX512F OR AVX10.1	Multiply the low double precision floating-point value in xmm3/m64 by low double precision floating-point value in xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies the low double precision floating-point value in the second source operand by the low double precision floating-point value in the first source operand, and stores the double precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source operand and the destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: The quadword at bits 127:64 of the destination operand is copied from the same bits of the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the write-mask.

Software should ensure VMULSD is encoded with VEX.L=0. Encoding VMULSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VMULSD (EVEX Encoded Version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := SRC1[63:0] * SRC2[63:0]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] := 0
            FI
    FI;
ENDFOR
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VMULSD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] * SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

MULSD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] * SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VMULSD __m128d _mm_mask_mul_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VMULSD __m128d _mm_maskz_mul_sd(__mmask8 k, __m128d a, __m128d b);
VMULSD __m128d _mm_mul_round_sd(__m128d a, __m128d b, int);
VMULSD __m128d _mm_mask_mul_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMULSD __m128d _mm_maskz_mul_round_sd(__mmask8 k, __m128d a, __m128d b, int);
MULSD __m128d _mm_mul_sd (__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-20, "Type 3 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-49, "Type E3 Class Exception Conditions."

MULSS—Multiply Scalar Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 59 /r MULSS xmm1,xmm2/m32	A	V/V	SSE	Multiply the low single precision floating-point value in xmm2/m32 by the low single precision floating-point value in xmm1.
VEX.LIG.F3.0F.WIG 59 /r VMULSS xmm1,xmm2, xmm3/m32	B	V/V	AVX	Multiply the low single precision floating-point value in xmm3/m32 by the low single precision floating-point value in xmm2.
EVEX.LLIG.F3.0F.W0 59 /r VMULSS xmm1 {k1}{z}, xmm2, xmm3/m32 {er}	C	V/V	AVX512F OR AVX10.1	Multiply the low single precision floating-point value in xmm3/m32 by the low single precision floating-point value in xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies the low single precision floating-point value from the second source operand by the low single precision floating-point value in the first source operand, and stores the single precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source operand and the destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The three high-order doublewords of the destination operand are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the write-mask.

Software should ensure VMULSS is encoded with VEX.L=0. Encoding VMULSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VMULSS (EVEX Encoded Version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := SRC1[31:0] * SRC2[31:0]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] := 0
            FI
    FI;
ENDFOR
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VMULSS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] * SRC2[31:0]
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

MULSS (128-bit Legacy SSE Version)

```
DEST[31:0] := DEST[31:0] * SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VMULSS __m128 _mm_mask_mul_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMULSS __m128 _mm_maskz_mul_ss(__mmask8 k, __m128 a, __m128 b);
VMULSS __m128 _mm_mul_round_ss(__m128 a, __m128 b, int);
VMULSS __m128 _mm_mask_mul_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMULSS __m128 _mm_maskz_mul_round_ss(__mmask8 k, __m128 a, __m128 b, int);
MULSS __m128 _mm_mul_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Underflow, Overflow, Invalid, Precision, Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-49, “Type E3 Class Exception Conditions.”

MULX—Unsigned Multiply Without Affecting Flags

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.LZ.F2.0F38.W0 F6 /r MULX r32a, r32b, r/m32	RVM	V/V	BMI2	Unsigned multiply of r/m32 with EDX without affecting arithmetic flags.
VEX.LZ.F2.0F38.W1 F6 /r MULX r64a, r64b, r/m64	RVM	V/N.E.	BMI2	Unsigned multiply of r/m64 with RDX without affecting arithmetic flags.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (w)	ModRM:r/m (r)	RDX/EDX is implied 64/32 bits source

Description

Performs an unsigned multiplication of the implicit source operand (EDX/RDX) and the specified source operand (the third operand) and stores the low half of the result in the second destination (second operand), the high half of the result in the first destination operand (first operand), without reading or writing the arithmetic flags. This enables efficient programming where the software can interleave add with carry operations and multiplications.

If the first and second operand are identical, it will contain the high half of the multiplication result.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
// DEST1: ModRM:reg
// DEST2: VEX.vvvv
IF (OperandSize = 32)
    SRC1 := EDX;
    DEST2 := (SRC1*SRC2)[31:0];
    DEST1 := (SRC1*SRC2)[63:32];
ELSE IF (OperandSize = 64)
    SRC1 := RDX;
        DEST2 := (SRC1*SRC2)[63:0];
        DEST1 := (SRC1*SRC2)[127:64];
FI
```

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);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-29, "Type 13 Class Exception Conditions."

MWAIT—Monitor Wait

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 C9	MWAIT	ZO	Valid	Valid	A hint that allows the processor to stop instruction execution and enter an implementation-dependent optimized state until occurrence of a class of events.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

The MWAIT instruction provides hints to allow the processor to enter an implementation-dependent optimized state. There are two principal targeted usages: address-range monitoring and advanced power management. The former usage requires the use of the MONITOR instruction, while the latter may or may not be combined with MONITOR.

CPUID.01H:ECX.MONITOR[3] and CPUID.07H.01H:EDX.MWAIT[23] both indicate the availability of MWAIT in the processor; the instruction is supported if either is enumerated as 1. System software may disable this instruction by clearing IA32_MISC_ENABLE[18]; doing so clears both CPUID feature flags and causes execution of MWAIT to generate an invalid-opcode exception.

MWAIT may be executed only at privilege level 0 (execution with CPL > 0 results in an invalid-opcode exception). The instruction's operation is the same in all modes.

ECX specifies extensions for the MWAIT instruction. EAX may contain hints such as the preferred optimized state the processor should enter. The first processors to implement MWAIT supported only the zero value for EAX and ECX. Later processors allowed setting ECX[0] to enable masked interrupts as break events for MWAIT or setting ECX[2] to enable monitorless MWAIT (see below). Software can use the CPUID instruction to determine the extensions supported by the processor:

- CPUID.05H:ECX.INTERRUPT_AS_BREAK_EVENT[1] indicates, if set, the ability to set ECX[0].
- CPUID.05H:ECX.MONITORLESS_MWAIT[3] indicates, if set, the ability to set ECX[2].

Execution of MWAIT with an unsupported extension selected in ECX results in a general-protection exception #GP(0).

Execution of MWAIT with ECX[2] = 1 is called "monitorless MWAIT." Such an execution ignores any preceding MONITOR instruction and stores to any address range that may have been monitored. Support for the MONITOR instruction is enumerated by CPUID.01H:ECX.MONITOR[3]. If CPUID.01H:ECX.MONITOR[3] is enumerated as 0 and CPUID.05H:ECX.MONITORLESS_MWAIT[3] is enumerated as 1, execution of MWAIT with ECX[2] = 0 results in #GP(0).

Address-range monitoring is always clear after MWAIT completes executing, even for executions of monitorless MWAIT.

MWAIT for Address Range Monitoring

For address-range monitoring, the MWAIT instruction operates with the MONITOR instruction. For that reason, it is available only if CPUID.01H:ECX.MONITOR[3] is enumerated as 1.

The two instructions allow the definition of an address at which to wait (MONITOR) and an implementation-dependent-optimized operation to commence at the wait address (MWAIT). The execution of MWAIT is a hint to the processor that it can enter an implementation-dependent-optimized state while waiting for an event or a store operation to the address range armed by MONITOR.

The following cause the processor to exit the implementation-dependent-optimized state: a store to the address range armed by the MONITOR instruction, an NMI or SMI, a debug exception, a machine check exception, the BINIT# signal, the INIT# signal, and the RESET# signal. Other implementation-dependent events may also cause the processor to exit the implementation-dependent-optimized state.

In addition, an external interrupt causes the processor to exit the implementation-dependent-optimized state either (1) if the interrupt would be delivered to software (e.g., as it would be if HLT had been executed instead of MWAIT); or (2) if ECX[0] = 1. Software can execute MWAIT with ECX[0] = 1 only if CPUID.05H:ECX.INTERRUPT_AS_BREAK_EVENT[1] = 1. (Implementation-specific conditions may result in an interrupt causing the processor to exit the implementation-dependent-optimized state even if interrupts are masked and ECX[0] = 0.)

Following exit from the implementation-dependent-optimized state, control passes to the instruction following the MWAIT instruction. A pending interrupt that is not masked (including an NMI or an SMI) may be delivered before execution of that instruction. Unlike the HLT instruction, the MWAIT instruction does not support a restart at the MWAIT instruction following the handling of an SMI.

If the preceding MONITOR instruction did not successfully arm an address range or if the MONITOR instruction has not been executed prior to executing MWAIT, then the processor will not enter the implementation-dependent-optimized state. Execution will resume at the instruction following the MWAIT.

MWAIT for Power Management

MWAIT accepts a hint and optional extension to the processor that it can enter a specified target C-state while waiting for an event or a store operation to the address range armed by MONITOR (the latter only if MWAIT is executed with ECX[2] = 0). Support for MWAIT extensions for power management is indicated by CPUID.05H:ECX[0] reporting 1.

EAX and ECX are used to communicate the additional information to the MWAIT instruction, such as the kind of optimized state the processor should enter. ECX specifies optional extensions for the MWAIT instruction. EAX may contain hints such as the preferred optimized state the processor should enter. Implementation-specific conditions may cause a processor to ignore the hint and enter a different optimized state. Future processor implementations may implement several optimized "waiting" states and will select among those states based on the hint argument.

Table 4-10 describes the meaning of ECX and EAX registers for MWAIT extensions.

Table 4-10. MWAIT Extension Register (ECX)

Bits	Description
0	Treat interrupts as break events even if masked (e.g., even if EFLAGS.IF = 0). May be set only if CPUID.05H:ECX.INTERRUPT_AS_BREAK_EVENT[1] = 1.
1	Reserved.
2	Allows MWAIT to enter and stay in an implementation-dependent-optimized state regardless of whether an address range armed by MONITOR exists or has been stored to. May be set only if CPUID.05H:ECX.MONITORLESS_MWAIT[3] = 1. Must be set if CPUID.01H:ECX.MONITOR[3] = 0.
31: 3	Reserved.

Table 4-11. MWAIT Hints Register (EAX)

Bits	Description
3 : 0	Sub C-state within a C-state, indicated by bits [7:4]
7 : 4	Target C-state* Value of 0 means C1; 1 means C2 and so on Value of 01111B means C0 Note: Target C states for MWAIT extensions are processor-specific C-states, not ACPI C-states
31: 8	Reserved

Note that if MWAIT is used to enter any of the C-states that are numerically higher than C1, a store to the address range armed by the MONITOR instruction will cause the processor to exit MWAIT only if the store was originated by other processor agents. A store from non-processor agent might not cause the processor to exit MWAIT in such cases.

For additional details of MWAIT extensions, see Chapter 17, "Power and Thermal Management," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3.

Operation

```
(* MWAIT takes the argument in EAX as a hint extension and is architected to take the argument in ECX as an instruction extension
MWAIT EAX, ECX *)
{
WHILE (("Monitor Hardware is in armed state") or ECX[2] = 1) {
    implementation_dependent_optimized_state(EAX, ECX); }
Set the state of Monitor Hardware as triggered;
}
```

Intel C/C++ Compiler Intrinsic Equivalent

MWAIT void _mm_mwait(unsigned extensions, unsigned hints)

Example Using a Monitored Address

MONITOR/MWAIT instruction pair must be coded in the same loop because execution of the MWAIT instruction will trigger the monitor hardware. It is not a proper usage to execute MONITOR once and then execute MWAIT in a loop. Setting up MONITOR without executing MWAIT has no adverse effects.

Typically the MONITOR/MWAIT pair is used in a sequence, such as:

```
EAX = Logical Address(Trigger)
ECX = 0 (*Hints *)
EDX = 0 (* Hints *)

IF ( !trigger_store_happened) {
    MONITOR EAX, ECX, EDX
    IF ( !trigger_store_happened ) {
        MWAIT EAX, ECX
    }
}
```

The above code sequence makes sure that a triggering store does not happen between the first check of the trigger and the execution of the monitor instruction. Without the second check that triggering store would go unnoticed. Typical usage of MONITOR and MWAIT would have the above code sequence within a loop.

Numeric Exceptions

None.

Protected Mode Exceptions

#GP(0)	If ECX[31:3] ≠ 0 or ECX[1] = 1.
	If ECX[0] = 1 and CPUID.05H:ECX.INTERRUPT_AS_BREAK_EVENT[1] = 0.
	If ECX[2] = 1 and CPUID.05H:ECX.MONITORLESS_MWAIT[3] = 0.
	If ECX[2] = 0 and CPUID.01H:ECX.MONITOR[3] = 0.
#UD	If CPUID.01H:ECX.MONITOR[3] = 0 and CPUID.07H.01H:EDX.MWAIT[23] = 0.
	If current privilege level is not 0.

Real Address Mode Exceptions

#GP(0)	If ECX[31:3] ≠ 0 or ECX[1] = 1.
	If ECX[0] = 1 and CPUID.05H:ECX.INTERRUPT_AS_BREAK_EVENT[1] = 0.
	If ECX[2] = 1 and CPUID.05H:ECX.MONITORLESS_MWAIT[3] = 0.
	If ECX[2] = 0 and CPUID.01H:ECX.MONITOR[3] = 0.

#UD	If CPUID.01H:ECX.MONITOR[3] = 0 and CPUID.07H.01H:EDX.MWAIT[23] = 0.

Virtual 8086 Mode Exceptions

#UD	The MWAIT instruction is not recognized in virtual-8086 mode (even if CPUID.01H:ECX.MONITOR[3] = 1 or CPUID.07H.01H:EDX.MWAIT[23] = 1).

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If RCX[63:3] ≠ 0 or RCX[1] = 1.
	If RCX[0] = 1 and CPUID.05H:ECX.INTERRUPT_AS_BREAK_EVENT[1] = 0.
	If RCX[2] = 1 and CPUID.05H:ECX.MONITORLESS_MWAIT[bit 3] = 0.
	If RCX[2] = 0 and CPUID.01H:ECX.MONITOR[bit 3] = 0.
#UD	If the current privilege level is not 0.
	If CPUID.01H:ECX.MONITOR[3] = 0 and CPUID.07H.01H:EDX.MWAIT[23] = 0.

NEG—Two's Complement Negation

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /3	NEG r/m8[1]	M	Valid	Valid	Two's complement negate r/m8.
F7 /3	NEG r/m16	M	Valid	Valid	Two's complement negate r/m16.
F7 /3	NEG r/m32	M	Valid	Valid	Two's complement negate r/m32.
REX.W + F7 /3	NEG r/m64	M	Valid	N.E.	Two's complement negate r/m64.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	N/A	N/A	N/A

Description

Replaces the value of operand (the destination operand) with its two's complement. (This operation is equivalent to subtracting the operand from 0.) The destination operand is located in a general-purpose register or a memory location.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF DEST = 0
    THEN CF := 0;
    ELSE CF := 1;
FI;
DEST := [- (DEST)]
```

Flags Affected

The CF flag set to 0 if the source operand is 0; otherwise it is set to 1. The OF, SF, ZF, AF, and PF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

NOP—No Operation

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 90	NOP	ZO	Valid	Valid	One byte no-operation instruction.
NP 0F 1F /0	NOP r/m16	M	Valid	Valid	Multi-byte no-operation instruction.
NP 0F 1F /0	NOP r/m32	M	Valid	Valid	Multi-byte no-operation instruction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A
M	ModRM:r/m (r)	N/A	N/A	N/A

Description

This instruction performs no operation. It is a one-byte or multi-byte NOP that takes up space in the instruction stream but does not impact machine context, except for the EIP register.

The multi-byte form of NOP is available on processors with model encoding:

- CPUID.01H:EAX[Bytes 11:8] = 0110B or 1111B

The multi-byte NOP instruction does not alter the content of a register and will not issue a memory operation. The instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

The one-byte NOP instruction is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.

The multi-byte NOP instruction performs no operation on supported processors and generates undefined opcode exception on processors that do not support the multi-byte NOP instruction.

The memory operand form of the instruction allows software to create a byte sequence of "no operation" as one instruction. For situations where multiple-byte NOPs are needed, the recommended operations (32-bit mode and 64-bit mode) are:

Table 4-12. Recommended Multi-Byte Sequence of NOP Instruction

Length	Assembly	Byte Sequence
2 bytes	66 NOP	66 90H
3 bytes	NOP DWORD ptr [EAX]	0F 1F 00H
4 bytes	NOP DWORD ptr [EAX + 00H]	0F 1F 40 00H
5 bytes	NOP DWORD ptr [EAX + EAX*1 + 00H]	0F 1F 44 00 00H
6 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00H]	66 0F 1F 44 00 00H
7 bytes	NOP DWORD ptr [EAX + 00000000H]	0F 1F 80 00 00 00 00H
8 bytes	NOP DWORD ptr [EAX + EAX*1 + 00000000H]	0F 1F 84 00 00 00 00 00H
9 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00000000H]	66 0F 1F 84 00 00 00 00 00H

Flags Affected

None.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

NOT—One's Complement Negation

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /2	NOT r/m8[1]	M	Valid	Valid	Reverse each bit of r/m8.
F7 /2	NOT r/m16	M	Valid	Valid	Reverse each bit of r/m16.
F7 /2	NOT r/m32	M	Valid	Valid	Reverse each bit of r/m32.
REX.W + F7 /2	NOT r/m64	M	Valid	N.E.	Reverse each bit of r/m64.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	N/A	N/A	N/A

Description

Performs a bitwise NOT operation (each 1 is set to 0, and each 0 is set to 1) on the destination operand and stores the result in the destination operand location. The destination operand can be a register or a memory location.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST := NOT DEST;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

OR—Logical Inclusive OR

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0C ib	OR AL, imm8	I	Valid	Valid	AL OR imm8.
0D iw	OR AX, imm16	I	Valid	Valid	AX OR imm16.
0D id	OR EAX, imm32	I	Valid	Valid	EAX OR imm32.
REX.W + 0D id	OR RAX, imm32	I	Valid	N.E.	RAX OR imm32 (sign-extended).
80 /1 ib	OR r/m8[1], imm8	MI	Valid	Valid	r/m8 OR imm8.
81 /1 iw	OR r/m16, imm16	MI	Valid	Valid	r/m16 OR imm16.
81 /1 id	OR r/m32, imm32	MI	Valid	Valid	r/m32 OR imm32.
REX.W + 81 /1 id	OR r/m64, imm32	MI	Valid	N.E.	r/m64 OR imm32 (sign-extended).
83 /1 ib	OR r/m16, imm8	MI	Valid	Valid	r/m16 OR imm8 (sign-extended).
83 /1 ib	OR r/m32, imm8	MI	Valid	Valid	r/m32 OR imm8 (sign-extended).
REX.W + 83 /1 ib	OR r/m64, imm8	MI	Valid	N.E.	r/m64 OR imm8 (sign-extended).
08 /r	OR r/m8[1], r8[1]	MR	Valid	Valid	r/m8 OR r8.
09 /r	OR r/m16, r16	MR	Valid	Valid	r/m16 OR r16.
09 /r	OR r/m32, r32	MR	Valid	Valid	r/m32 OR r32.
REX.W + 09 /r	OR r/m64, r64	MR	Valid	N.E.	r/m64 OR r64.
0A /r	OR r8[1], r/m8[1]	RM	Valid	Valid	r8 OR r/m8.
0B /r	OR r16, r/m16	RM	Valid	Valid	r16 OR r/m16.
0B /r	OR r32, r/m32	RM	Valid	Valid	r32 OR r/m32.
REX.W + 0B /r	OR r64, r/m64	RM	Valid	N.E.	r64 OR r/m64.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	imm8/16/32	N/A	N/A
MI	ModRM:r/m (r, w)	imm8/16/32	N/A	N/A
MR	ModRM:r/m (r, w)	ModRM:reg (r)	N/A	N/A
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A

Description

Performs a bitwise inclusive OR operation between the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result of the OR instruction is set to 0 if both corresponding bits of the first and second operands are 0; otherwise, each bit is set to 1.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST := DEST OR SRC;
```

Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

ORPD—Bitwise Logical OR of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 56/r ORPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical OR of packed double precision floating-point values in xmm1 and xmm2/mem.
VEX.128.66.0F 56 /r VORPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical OR of packed double precision floating-point values in xmm2 and xmm3/mem.
VEX.256.66.0F 56 /r VORPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical OR of packed double precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.66.0F.W1 56 /r VORPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical OR of packed double precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.256.66.0F.W1 56 /r VORPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical OR of packed double precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.512.66.0F.W1 56 /r VORPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ OR AVX10.1	Return the bitwise logical OR of packed double precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical OR of the two, four or eight packed double precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with write-mask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VORPD (EVEX Encoded Versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := SRC1[i+63:i] BITWISE OR SRC2[63:0]
                ELSE
                    DEST[i+63:i] := SRC1[i+63:i] BITWISE OR SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VORPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[127:64] := SRC1[127:64] BITWISE OR SRC2[127:64]
DEST[191:128] := SRC1[191:128] BITWISE OR SRC2[191:128]
DEST[255:192] := SRC1[255:192] BITWISE OR SRC2[255:192]
DEST[MAXVL-1:256] := 0
```

VORPD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[127:64] := SRC1[127:64] BITWISE OR SRC2[127:64]
DEST[MAXVL-1:128] := 0
```

ORPD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] BITWISE OR SRC[63:0]
DEST[127:64] := DEST[127:64] BITWISE OR SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VORPD __m512d _mm512_or_pd ( __m512d a, __m512d  b);
VORPD __m512d _mm512_mask_or_pd ( __m512d s, __mmask8 k, __m512d a, __m512d b);
VORPD __m512d _mm512_maskz_or_pd (__mmask8 k, __m512d a, __m512d b);
VORPD __m256d _mm256_mask_or_pd (__m256d s, ___mmask8 k, __m256d a, __m256d b);
VORPD __m256d _mm256_maskz_or_pd (__mmask8 k, __m256d a, __m256d b);
VORPD __m128d _mm_mask_or_pd ( __m128d s, __mmask8 k, __m128d a, __m128d b);
VORPD __m128d _mm_maskz_or_pd (__mmask8 k, __m128d a, __m128d b);
VORPD __m256d _mm256_or_pd (__m256d a, __m256d b);
ORPD __m128d _mm_or_pd (__m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

ORPS—Bitwise Logical OR of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 56 /r ORPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical OR of packed single precision floating-point values in xmm1 and xmm2/mem.
VEX.128.0F 56 /r VORPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical OR of packed single precision floating-point values in xmm2 and xmm3/mem.
VEX.256.0F 56 /r VORPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical OR of packed single precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.0F.W0 56 /r VORPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical OR of packed single precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.256.0F.W0 56 /r VORPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical OR of packed single precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.512.0F.W0 56 /r VORPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512DQ OR AVX10.1	Return the bitwise logical OR of packed single precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical OR of the four, eight or sixteen packed single precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with write-mask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VORPS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] := SRC1[i+31:i] BITWISE OR SRC2[31:0]
                ELSE
                    DEST[i+31:i] := SRC1[i+31:i] BITWISE OR SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VORPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] := SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] := SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] := SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[159:128] := SRC1[159:128] BITWISE OR SRC2[159:128]
DEST[191:160] := SRC1[191:160] BITWISE OR SRC2[191:160]
DEST[223:192] := SRC1[223:192] BITWISE OR SRC2[223:192]
DEST[255:224] := SRC1[255:224] BITWISE OR SRC2[255:224].
DEST[MAXVL-1:256] := 0
```

VORPS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] := SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] := SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] := SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[MAXVL-1:128] := 0
```

ORPS (128-bit Legacy SSE Version)

```
DEST[31:0] := SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] := SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] := SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] := SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VORPS __m512 _mm512_or_ps (__m512 a, __m512 b);
VORPS __m512 _mm512_mask_or_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VORPS __m512 _mm512_maskz_or_ps (__mmask16 k, __m512 a, __m512 b);
VORPS __m256 _mm256_mask_or_ps (__m256 s, ___mmask8 k, __m256 a, __m256 b);
VORPS __m256 _mm256_maskz_or_ps (__mmask8 k, __m256 a, __m256 b);
VORPS __m128 _mm_mask_or_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VORPS __m128 _mm_maskz_or_ps (__mmask8 k, __m128 a, __m128 b);
VORPS __m256 _mm256_or_ps (__m256 a, __m256 b);
ORPS __m128 _mm_or_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."

EVEX-encoded instruction, see Table 2-51, "Type E4 Class Exception Conditions."

OUT—Output to Port

Opcode[1]	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
E6 ib	OUT imm8, AL	I	Valid	Valid	Output byte in AL to I/O port address imm8.
E7 ib	OUT imm8, AX	I	Valid	Valid	Output word in AX to I/O port address imm8.
E7 ib	OUT imm8, EAX	I	Valid	Valid	Output doubleword in EAX to I/O port address imm8.
EE	OUT DX, AL	ZO	Valid	Valid	Output byte in AL to I/O port address in DX.
EF	OUT DX, AX	ZO	Valid	Valid	Output word in AX to I/O port address in DX.
EF	OUT DX, EAX	ZO	Valid	Valid	Output doubleword in EAX to I/O port address in DX.

NOTES:

1. See the IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	imm8	N/A	N/A	N/A
ZO	N/A	N/A	N/A	N/A

Description

Copies the value from the second operand (source operand) to the I/O port specified with the destination operand (first operand). The source operand can be register AL, AX, or EAX, depending on the size of the port being accessed (8, 16, or 32 bits, respectively); the destination operand can be a byte-immediate or the DX register. Using a byte immediate allows I/O port addresses 0 to 255 to be accessed; using the DX register as a source operand allows I/O ports from 0 to 65,535 to be accessed.

The size of the I/O port being accessed is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

At the machine code level, I/O instructions are shorter when accessing 8-bit I/O ports. Here, the upper eight bits of the port address will be 0.

This instruction is only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 20, "Input/Output," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for more information on accessing I/O ports in the I/O address space.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility

After executing an OUT instruction, the Pentium® processor ensures that the EWBE# pin has been sampled active before it begins to execute the next instruction. (Note that the instruction can be prefetched if EWBE# is not active, but it will not be executed until the EWBE# pin is sampled active.) Only the Pentium processor family has the EWBE# pin.

Operation

```
IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
    THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
        IF (Any I/O Permission Bit for I/O port being accessed = 1)
            THEN (* I/O operation is not allowed *)
                #GP(0);
            ELSE ( * I/O operation is allowed *)
                DEST := SRC; (* Writes to selected I/O port *)
        FI;
    ELSE (Real Mode or Protected Mode with CPL ≤ IOPL *)
        DEST := SRC; (* Writes to selected I/O port *)
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as protected mode exceptions.

64-Bit Mode Exceptions

Same as protected mode exceptions.

OUTS/OUTSB/OUTSW/OUTSD—Output String to Port

Opcode[1]	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
6E	OUTS DX, m8	ZO	Valid	Valid	Output byte from memory location specified in DS:(E)SI or RSI to I/O port specified in DX[2].
6F	OUTS DX, m16	ZO	Valid	Valid	Output word from memory location specified in DS:(E)SI or RSI to I/O port specified in DX[2].
6F	OUTS DX, m32	ZO	Valid	Valid	Output doubleword from memory location specified in DS:(E)SI or RSI to I/O port specified in DX[2].
6E	OUTSB	ZO	Valid	Valid	Output byte from memory location specified in DS:(E)SI or RSI to I/O port specified in DX[2].
6F	OUTSW	ZO	Valid	Valid	Output word from memory location specified in DS:(E)SI or RSI to I/O port specified in DX[2].
6F	OUTSD	ZO	Valid	Valid	Output doubleword from memory location specified in DS:(E)SI or RSI to I/O port specified in DX[2].

NOTES:

1. See the IA-32 Architecture Compatibility section below.
2. In 64-bit mode, only 64-bit (RSI) and 32-bit (ESI) address sizes are supported. In non-64-bit mode, only 32-bit (ESI) and 16-bit (SI) address sizes are supported.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Copies data from the source operand (second operand) to the I/O port specified with the destination operand (first operand). The source operand is a memory location, the address of which is read from either the DS:SI, DS:ESI or the RSI registers (depending on the address-size attribute of the instruction, 16, 32 or 64, respectively). (The DS segment may be overridden with a segment override prefix.) The destination operand is an I/O port address (from 0 to 65,535) that is read from the DX register. The size of the I/O port being accessed (that is, the size of the source and destination operands) is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operands" form and the "no-operands" form. The explicit-operands form (specified with the OUTS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source operand should be a symbol that indicates the size of the I/O port and the source address, and the destination operand must be DX. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the DS:(E)SI or RSI registers, which must be loaded correctly before the OUTS instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the OUTS instructions. Here also DS:(E)SI is assumed to be the source operand and DX is assumed to be the destination operand. The size of the I/O port is specified with the choice of mnemonic: OUTSB (byte), OUTSW (word), or OUTSD (doubleword).

After the byte, word, or doubleword is transferred from the memory location to the I/O port, the SI/ESI/RSI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI register is incremented; if the DF flag is 1, the SI/ESI/RSI register is decremented.) The SI/ESI/RSI register is incremented or decremented by 1 for byte operations, by 2 for word operations, and by 4 for doubleword operations.

The OUTS, OUTSB, OUTSW, and OUTSD instructions can be preceded by the REP prefix for block input of ECX bytes, words, or doublewords. See "REP—Repeat String Operation (Prefix)" in this chapter for a description of the

REP prefix. This instruction is only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 20, "Input/Output," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for more information on accessing I/O ports in the I/O address space.

In 64-bit mode, the default operand size is 32 bits; operand size is not promoted by the use of REX.W. In 64-bit mode, the default address size is 64 bits, and 64-bit address is specified using RSI by default. 32-bit address using ESI is support using the prefix 67H, but 16-bit address is not supported in 64-bit mode.

IA-32 Architecture Compatibility

After executing an OUTS, OUTSB, OUTSW, or OUTSD instruction, the Pentium processor ensures that the EWBE# pin has been sampled active before it begins to execute the next instruction. (Note that the instruction can be prefetched if EWBE# is not active, but it will not be executed until the EWBE# pin is sampled active.) Only the Pentium processor family has the EWBE# pin.

For the Pentium 4, Intel® Xeon®, and P6 processor family, upon execution of an OUTS, OUTSB, OUTSW, or OUTSD instruction, the processor will not execute the next instruction until the data phase of the transaction is complete.

Operation

```
IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
    THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
        IF (Any I/O Permission Bit for I/O port being accessed = 1)
            THEN (* I/O operation is not allowed *)
                #GP(0);
            ELSE (* I/O operation is allowed *)
                DEST := SRC; (* Writes to I/O port *)
        FI;
    ELSE (Real Mode or Protected Mode or 64-Bit Mode with CPL ≤ IOPL *)
        DEST := SRC; (* Writes to I/O port *)
FI;

Byte transfer:
    IF 64-bit mode
        Then
            IF 64-Bit Address Size
                THEN
                    IF DF = 0
                        THEN RSI := RSI RSI + 1;
                        ELSE RSI := RSI or – 1;
                    FI;
                ELSE (* 32-Bit Address Size *)
                    IF DF = 0
                        THEN     ESI := ESI + 1;
                        ELSE     ESI := ESI – 1;
                    FI;
            FI;
        ELSE
            IF DF = 0
                THEN     (E)SI := (E)SI + 1;
                ELSE (E)SI := (E)SI – 1;
            FI;
    FI;
Word transfer:
    IF 64-bit mode
        Then
            IF 64-Bit Address Size
```

```
                THEN
                    IF DF = 0
                        THEN RSI := RSI RSI + 2;
                        ELSE RSI := RSI or – 2;
                    FI;
                ELSE (* 32-Bit Address Size *)
                    IF DF = 0
                        THEN    ESI := ESI + 2;
                        ELSE    ESI := ESI – 2;
                    FI;
            FI;
        ELSE
            IF DF = 0
                THEN    (E)SI := (E)SI + 2;
                ELSE (E)SI := (E)SI – 2;
            FI;
    FI;
Doubleword transfer:
    IF 64-bit mode
        Then
            IF 64-Bit Address Size
                THEN
                    IF DF = 0
                        THEN RSI := RSI RSI + 4;
                        ELSE RSI := RSI or – 4;
                    FI;
                ELSE (* 32-Bit Address Size *)
                    IF DF = 0
                        THEN    ESI := ESI + 4;
                        ELSE    ESI := ESI – 4;
                    FI;
            FI;
        ELSE
            IF DF = 0
                THEN    (E)SI := (E)SI + 4;
                ELSE (E)SI := (E)SI – 4;
            FI;
    FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1. If a memory operand effective address is outside the limit of the CS, DS, ES, FS, or GS segment. If the segment register contains a NULL segment selector.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

PABSB/PABSW/PABSD/PABSQ—Packed Absolute Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 1C /r[1] PABSB mm1, mm2/m64	A	V/V	SSSE3	Compute the absolute value of bytes in mm2/m64 and store UNSIGNED result in mm1.
66 0F 38 1C /r PABSB xmm1, xmm2/m128	A	V/V	SSSE3	Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1.
NP 0F 38 1D /r[1] PABSW mm1, mm2/m64	A	V/V	SSSE3	Compute the absolute value of 16-bit integers in mm2/m64 and store UNSIGNED result in mm1.
66 0F 38 1D /r PABSW xmm1, xmm2/m128	A	V/V	SSSE3	Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
NP 0F 38 1E /r[1] PABSD mm1, mm2/m64	A	V/V	SSSE3	Compute the absolute value of 32-bit integers in mm2/m64 and store UNSIGNED result in mm1.
66 0F 38 1E /r PABSD xmm1, xmm2/m128	A	V/V	SSSE3	Compute the absolute value of 32-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
VEX.128.66.0F38.WIG 1C /r VPABSB xmm1, xmm2/m128	A	V/V	AVX	Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1.
VEX.128.66.0F38.WIG 1D /r VPABSW xmm1, xmm2/m128	A	V/V	AVX	Compute the absolute value of 16- bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
VEX.128.66.0F38.WIG 1E /r VPABSD xmm1, xmm2/m128	A	V/V	AVX	Compute the absolute value of 32- bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
VEX.256.66.0F38.WIG 1C /r VPABSB ymm1, ymm2/m256	A	V/V	AVX2	Compute the absolute value of bytes in ymm2/m256 and store UNSIGNED result in ymm1.
VEX.256.66.0F38.WIG 1D /r VPABSW ymm1, ymm2/m256	A	V/V	AVX2	Compute the absolute value of 16-bit integers in ymm2/m256 and store UNSIGNED result in ymm1.
VEX.256.66.0F38.WIG 1E /r VPABSD ymm1, ymm2/m256	A	V/V	AVX2	Compute the absolute value of 32-bit integers in ymm2/m256 and store UNSIGNED result in ymm1.
EVEX.128.66.0F38.WIG 1C /r VPABSB xmm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1 using writemask k1.
EVEX.256.66.0F38.WIG 1C /r VPABSB ymm1 {k1}{z}, ymm2/m256	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compute the absolute value of bytes in ymm2/m256 and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.WIG 1C /r VPABSB zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512BW OR AVX10.1	Compute the absolute value of bytes in zmm2/m512 and store UNSIGNED result in zmm1 using writemask k1.
EVEX.128.66.0F38.WIG 1D /r VPABSW xmm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1 using writemask k1.
EVEX.256.66.0F38.WIG 1D /r VPABSW ymm1 {k1}{z}, ymm2/m256	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compute the absolute value of 16-bit integers in ymm2/m256 and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.WIG 1D /r VPABSW zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512BW OR AVX10.1	Compute the absolute value of 16-bit integers in zmm2/m512 and store UNSIGNED result in zmm1 using writemask k1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 1E /r VPABSD xmm1 {k1}{z}, xmm2/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compute the absolute value of 32-bit integers in xmm2/m128/m32bcst and store UNSIGNED result in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 1E /r VPABSD ymm1 {k1}{z}, ymm2/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compute the absolute value of 32-bit integers in ymm2/m256/m32bcst and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 1E /r VPABSD zmm1 {k1}{z}, zmm2/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Compute the absolute value of 32-bit integers in zmm2/m512/m32bcst and store UNSIGNED result in zmm1 using writemask k1.
EVEX.128.66.0F38.W1 1F /r VPABSQ xmm1 {k1}{z}, xmm2/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compute the absolute value of 64-bit integers in xmm2/m128/m64bcst and store UNSIGNED result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 1F /r VPABSQ ymm1 {k1}{z}, ymm2/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compute the absolute value of 64-bit integers in ymm2/m256/m64bcst and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 1F /r VPABSQ zmm1 {k1}{z}, zmm2/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Compute the absolute value of 64-bit integers in zmm2/m512/m64bcst and store UNSIGNED result in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
C	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

PABSB/W/D computes the absolute value of each data element of the source operand (the second operand) and stores the UNSIGNED results in the destination operand (the first operand). PABSB operates on signed bytes, PABSW operates on signed 16-bit words, and PABSD operates on signed 32-bit integers.

EVEX encoded VPABSD/Q: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

EVEX encoded VPABSB/W: The source operand is a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded versions: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding register destination are zeroed.

VEX.128 encoded versions: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The source operand can be an XMM register or an 128-bit memory location. The destination is an XMM register. The upper bits (VL_MAX-1:128) of the corresponding register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

PABSB With 64-bit Operands:

Unsigned DEST[7:0] := ABS(SRC[7: 0])
Repeat operation for 2nd through 7th bytes
Unsigned DEST[63:56] := ABS(SRC[63:56])

PABSB With 128-bit Operands:

Unsigned DEST[7:0] := ABS(SRC[7: 0])
Repeat operation for 2nd through 15th bytes
Unsigned DEST[127:120] := ABS(SRC[127:120])

VPABSB With 128-bit Operands:

Unsigned DEST[7:0] := ABS(SRC[7: 0])
Repeat operation for 2nd through 15th bytes
Unsigned DEST[127:120] := ABS(SRC[127:120])

VPABSB With 256-bit Operands:

Unsigned DEST[7:0] := ABS(SRC[7: 0])
Repeat operation for 2nd through 31st bytes
Unsigned DEST[255:248] := ABS(SRC[255:248])

VPABSB (EVEX Encoded Versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

```
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN
            Unsigned DEST[i+7:i] := ABS(SRC[i+7:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

PABSW With 128-bit Operands:

Unsigned DEST[15:0] := ABS(SRC[15:0])
Repeat operation for 2nd through 7th 16-bit words
Unsigned DEST[127:112] := ABS(SRC[127:112])

VPABSW With 128-bit Operands:

Unsigned DEST[15:0] := ABS(SRC[15:0])
Repeat operation for 2nd through 7th 16-bit words
Unsigned DEST[127:112] := ABS(SRC[127:112])

VPABSW With 256-bit Operands:

Unsigned DEST[15:0] := ABS(SRC[15:0])
Repeat operation for 2nd through 15th 16-bit words
Unsigned DEST[255:240] := ABS(SRC[255:240])

VPABSW (EVEX Encoded Versions)

```
    (KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN
            Unsigned DEST[i+15:i] := ABS(SRC[i+15:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

PABSD With 128-bit Operands:

```
    Unsigned DEST[31:0] := ABS(SRC[31:0])
    Repeat operation for 2nd through 3rd 32-bit double words
    Unsigned DEST[127:96] := ABS(SRC[127:96])
```

VPABSD With 128-bit Operands:

```
    Unsigned DEST[31:0] := ABS(SRC[31:0])
    Repeat operation for 2nd through 3rd 32-bit double words
    Unsigned DEST[127:96] := ABS(SRC[127:96])
```

VPABSD With 256-bit Operands:

```
    Unsigned DEST[31:0] := ABS(SRC[31:0])
    Repeat operation for 2nd through 7th 32-bit double words
    Unsigned DEST[255:224] := ABS(SRC[255:224])
```

VPABSD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN
                    Unsigned DEST[i+31:i] := ABS(SRC[31:0])
                ELSE
                    Unsigned DEST[i+31:i] := ABS(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR;
```

```
DEST[MAXVL-1:VL] := 0
```

VPABSQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN
                    Unsigned DEST[i+63:i] := ABS(SRC[63:0])
                ELSE
                    Unsigned DEST[i+63:i] := ABS(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPABSB __m512i _mm512_abs_epi8 ( __m512i a)
VPABSW __m512i _mm512_abs_epi16 ( __m512i a)
VPABSB __m512i _mm512_mask_abs_epi8 ( __m512i s, __mmask64 m, __m512i a)
VPABSW __m512i _mm512_mask_abs_epi16 ( __m512i s, __mmask32 m, __m512i a)
VPABSB __m512i _mm512_maskz_abs_epi8 (__mmask64 m, __m512i a)
VPABSW __m512i _mm512_maskz_abs_epi16 (__mmask32 m, __m512i a)
VPABSB __m256i _mm256_mask_abs_epi8 (__m256i s, __mmask32 m, __m256i a)
VPABSW __m256i _mm256_mask_abs_epi16 (__m256i s, __mmask16 m, __m256i a)
VPABSB __m256i _mm256_maskz_abs_epi8 (__mmask32 m, __m256i a)
VPABSW __m256i _mm256_maskz_abs_epi16 (__mmask16 m, __m256i a)
VPABSB __m128i _mm_mask_abs_epi8 (__m128i s, __mmask16 m, __m128i a)
VPABSW __m128i _mm_mask_abs_epi16 (__m128i s, __mmask8 m, __m128i a)
VPABSB __m128i _mm_maskz_abs_epi8 (__mmask16 m, __m128i a)
VPABSW __m128i _mm_maskz_abs_epi16 (__mmask8 m, __m128i a)
VPABSD __m256i _mm256_mask_abs_epi32(__m256i s, __mmask8 k, __m256i a);
VPABSD __m256i _mm256_maskz_abs_epi32( __mmask8 k, __m256i a);
VPABSD __m128i _mm_mask_abs_epi32(__m128i s, __mmask8 k, __m128i a);
VPABSD __m128i _mm_maskz_abs_epi32( __mmask8 k, __m128i a);
VPABSD  __m512i _mm512_abs_epi32( __m512i a);
VPABSD  __m512i _mm512_mask_abs_epi32(__m512i s, __mmask16 k, __m512i a);
VPABSD  __m512i _mm512_maskz_abs_epi32( __mmask16 k, __m512i a);
VPABSQ __m512i _mm512_abs_epi64( __m512i a);
VPABSQ __m512i _mm512_mask_abs_epi64(__m512i s, __mmask8 k, __m512i a);
VPABSQ __m512i _mm512_maskz_abs_epi64( __mmask8 k, __m512i a);
VPABSQ __m256i _mm256_mask_abs_epi64(__m256i s, __mmask8 k, __m256i a);
VPABSQ __m256i _mm256_maskz_abs_epi64( __mmask8 k, __m256i a);
VPABSQ __m128i _mm_mask_abs_epi64(__m128i s, __mmask8 k, __m128i a);
VPABSQ __m128i _mm_maskz_abs_epi64( __mmask8 k, __m128i a);
```

PABSB __m128i _mm_abs_epi8 (__m128i a)
VPABSB __m128i _mm_abs_epi8 (__m128i a)
VPABSB __m256i _mm256_abs_epi8 (__m256i a)
PABSW __m128i _mm_abs_epi16 (__m128i a)
VPABSW __m128i _mm_abs_epi16 (__m128i a)
VPABSW __m256i _mm256_abs_epi16 (__m256i a)
PABSD __m128i _mm_abs_epi32 (__m128i a)
VPABSD __m128i _mm_abs_epi32 (__m128i a)
VPABSD __m256i _mm256_abs_epi32 (__m256i a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded VPABSD/Q, see Table 2-51, "Type E4 Class Exception Conditions."
EVEX-encoded VPABSB/W, see Exceptions Type E4.nb in Table 2-51, "Type E4 Class Exception Conditions."

PACKSSWB/PACKSSDW—Pack With Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 63 /r[1] PACKSSWB mm1, mm2/m64	A	V/V	MMX	Converts 4 packed signed word integers from mm1 and from mm2/m64 into 8 packed signed byte integers in mm1 using signed saturation.
66 0F 63 /r PACKSSWB xmm1, xmm2/m128	A	V/V	SSE2	Converts 8 packed signed word integers from xmm1 and from xmm2/m128 into 16 packed signed byte integers in xmm1 using signed saturation.
NP 0F 6B /r[1] PACKSSDW mm1, mm2/m64	A	V/V	MMX	Converts 2 packed signed doubleword integers from mm1 and from mm2/m64 into 4 packed signed word integers in mm1 using signed saturation.
66 0F 6B /r PACKSSDW xmm1, xmm2/m128	A	V/V	SSE2	Converts 4 packed signed doubleword integers from xmm1 and from xmm2/m128 into 8 packed signed word integers in xmm1 using signed saturation.
VEX.128.66.0F.WIG 63 /r VPACKSSWB xmm1,xmm2, xmm3/m128	B	V/V	AVX	Converts 8 packed signed word integers from xmm2 and from xmm3/m128 into 16 packed signed byte integers in xmm1 using signed saturation.
VEX.128.66.0F.WIG 6B /r VPACKSSDW xmm1,xmm2, xmm3/m128	B	V/V	AVX	Converts 4 packed signed doubleword integers from xmm2 and from xmm3/m128 into 8 packed signed word integers in xmm1 using signed saturation.
VEX.256.66.0F.WIG 63 /r VPACKSSWB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Converts 16 packed signed word integers from ymm2 and from ymm3/m256 into 32 packed signed byte integers in ymm1 using signed saturation.
VEX.256.66.0F.WIG 6B /r VPACKSSDW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Converts 8 packed signed doubleword integers from ymm2 and from ymm3/m256 into 16 packed signed word integers in ymm1using signed saturation.
EVEX.128.66.0F.WIG 63 /r VPACKSSWB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Converts packed signed word integers from xmm2 and from xmm3/m128 into packed signed byte integers in xmm1 using signed saturation under writemask k1.
EVEX.256.66.0F.WIG 63 /r VPACKSSWB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Converts packed signed word integers from ymm2 and from ymm3/m256 into packed signed byte integers in ymm1 using signed saturation under writemask k1.
EVEX.512.66.0F.WIG 63 /r VPACKSSWB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Converts packed signed word integers from zmm2 and from zmm3/m512 into packed signed byte integers in zmm1 using signed saturation under writemask k1.
EVEX.128.66.0F.W0 6B /r VPACKSSDW xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Converts packed signed doubleword integers from xmm2 and from xmm3/m128/m32bcst into packed signed word integers in xmm1 using signed saturation under writemask k1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F.W0 6B /r VPACKSSDW ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Converts packed signed doubleword integers from ymm2 and from ymm3/m256/m32bcst into packed signed word integers in ymm1 using signed saturation under writemask k1.
EVEX.512.66.0F.W0 6B /r VPACKSSDW zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512BW OR AVX10.1	Converts packed signed doubleword integers from zmm2 and from zmm3/m512/m32bcst into packed signed word integers in zmm1 using signed saturation under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Converts packed signed word integers into packed signed byte integers (PACKSSWB) or converts packed signed doubleword integers into packed signed word integers (PACKSSDW), using saturation to handle overflow conditions. See Figure 4-6 for an example of the packing operation.

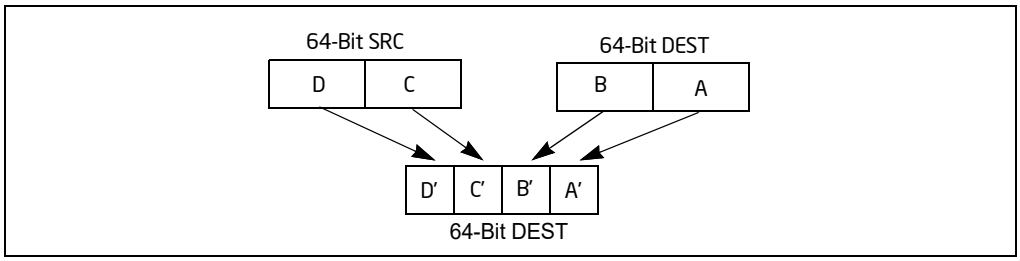


Figure 4-6. Operation of the PACKSSDW Instruction Using 64-Bit Operands

PACKSSWB converts packed signed word integers in the first and second source operands into packed signed byte integers using signed saturation to handle overflow conditions beyond the range of signed byte integers. If the signed word value is beyond the range of a signed byte value (i.e., greater than 7FH or less than 80H), the saturated signed byte integer value of 7FH or 80H, respectively, is stored in the destination. PACKSSDW converts packed signed doubleword integers in the first and second source operands into packed signed word integers using signed saturation to handle overflow conditions beyond 7FFFH and 8000H.

EVEX encoded PACKSSWB: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register, updated conditional under the writemask k1.

EVEX encoded PACKSSDW: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-

bit memory location. The destination operand is a ZMM/YMM/XMM register, updated conditional under the write-mask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM destination register destination are unmodified.

Operation

PACKSSWB Instruction (128-bit Legacy SSE Version)

```
DEST[7:0] := SaturateSignedWordToSignedByte (DEST[15:0]);
DEST[15:8] := SaturateSignedWordToSignedByte (DEST[31:16]);
DEST[23:16] := SaturateSignedWordToSignedByte (DEST[47:32]);
DEST[31:24] := SaturateSignedWordToSignedByte (DEST[63:48]);
DEST[39:32] := SaturateSignedWordToSignedByte (DEST[79:64]);
DEST[47:40] := SaturateSignedWordToSignedByte (DEST[95:80]);
DEST[55:48] := SaturateSignedWordToSignedByte (DEST[111:96]);
DEST[63:56] := SaturateSignedWordToSignedByte (DEST[127:112]);
DEST[71:64] := SaturateSignedWordToSignedByte (SRC[15:0]);
DEST[79:72] := SaturateSignedWordToSignedByte (SRC[31:16]);
DEST[87:80] := SaturateSignedWordToSignedByte (SRC[47:32]);
DEST[95:88] := SaturateSignedWordToSignedByte (SRC[63:48]);
DEST[103:96] := SaturateSignedWordToSignedByte (SRC[79:64]);
DEST[111:104] := SaturateSignedWordToSignedByte (SRC[95:80]);
DEST[119:112] := SaturateSignedWordToSignedByte (SRC[111:96]);
DEST[127:120] := SaturateSignedWordToSignedByte (SRC[127:112]);
DEST[MAXVL-1:128] (Unmodified)
```

PACKSSDW Instruction (128-bit Legacy SSE Version)

```
DEST[15:0] := SaturateSignedDwordToSignedWord (DEST[31:0]);
DEST[31:16] := SaturateSignedDwordToSignedWord (DEST[63:32]);
DEST[47:32] := SaturateSignedDwordToSignedWord (DEST[95:64]);
DEST[63:48] := SaturateSignedDwordToSignedWord (DEST[127:96]);
DEST[79:64] := SaturateSignedDwordToSignedWord (SRC[31:0]);
DEST[95:80] := SaturateSignedDwordToSignedWord (SRC[63:32]);
DEST[111:96] := SaturateSignedDwordToSignedWord (SRC[95:64]);
DEST[127:112] := SaturateSignedDwordToSignedWord (SRC[127:96]);
DEST[MAXVL-1:128] (Unmodified)
```

VPACKSSWB Instruction (VEX.128 Encoded Version)

```
DEST[7:0] := SaturateSignedWordToSignedByte (SRC1[15:0]);
DEST[15:8] := SaturateSignedWordToSignedByte (SRC1[31:16]);
DEST[23:16] := SaturateSignedWordToSignedByte (SRC1[47:32]);
DEST[31:24] := SaturateSignedWordToSignedByte (SRC1[63:48]);
DEST[39:32] := SaturateSignedWordToSignedByte (SRC1[79:64]);
DEST[47:40] := SaturateSignedWordToSignedByte (SRC1[95:80]);
DEST[55:48] := SaturateSignedWordToSignedByte (SRC1[111:96]);
DEST[63:56] := SaturateSignedWordToSignedByte (SRC1[127:112]);
DEST[71:64] := SaturateSignedWordToSignedByte (SRC2[15:0]);
DEST[79:72] := SaturateSignedWordToSignedByte (SRC2[31:16]);
DEST[87:80] := SaturateSignedWordToSignedByte (SRC2[47:32]);
DEST[95:88] := SaturateSignedWordToSignedByte (SRC2[63:48]);
DEST[103:96] := SaturateSignedWordToSignedByte (SRC2[79:64]);
DEST[111:104] := SaturateSignedWordToSignedByte (SRC2[95:80]);
DEST[119:112] := SaturateSignedWordToSignedByte (SRC2[111:96]);
DEST[127:120] := SaturateSignedWordToSignedByte (SRC2[127:112]);
DEST[MAXVL-1:128] := 0;
```

VPACKSSDW Instruction (VEX.128 Encoded Version)

```
DEST[15:0] := SaturateSignedDwordToSignedWord (SRC1[31:0]);
DEST[31:16] := SaturateSignedDwordToSignedWord (SRC1[63:32]);
DEST[47:32] := SaturateSignedDwordToSignedWord (SRC1[95:64]);
DEST[63:48] := SaturateSignedDwordToSignedWord (SRC1[127:96]);
DEST[79:64] := SaturateSignedDwordToSignedWord (SRC2[31:0]);
DEST[95:80] := SaturateSignedDwordToSignedWord (SRC2[63:32]);
DEST[111:96] := SaturateSignedDwordToSignedWord (SRC2[95:64]);
DEST[127:112] := SaturateSignedDwordToSignedWord (SRC2[127:96]);
DEST[MAXVL-1:128] := 0;
```

VPACKSSWB Instruction (VEX.256 Encoded Version)

```
DEST[7:0] := SaturateSignedWordToSignedByte (SRC1[15:0]);
DEST[15:8] := SaturateSignedWordToSignedByte (SRC1[31:16]);
DEST[23:16] := SaturateSignedWordToSignedByte (SRC1[47:32]);
DEST[31:24] := SaturateSignedWordToSignedByte (SRC1[63:48]);
DEST[39:32] := SaturateSignedWordToSignedByte (SRC1[79:64]);
DEST[47:40] := SaturateSignedWordToSignedByte (SRC1[95:80]);
DEST[55:48] := SaturateSignedWordToSignedByte (SRC1[111:96]);
DEST[63:56] := SaturateSignedWordToSignedByte (SRC1[127:112]);
DEST[71:64] := SaturateSignedWordToSignedByte (SRC2[15:0]);
DEST[79:72] := SaturateSignedWordToSignedByte (SRC2[31:16]);
DEST[87:80] := SaturateSignedWordToSignedByte (SRC2[47:32]);
DEST[95:88] := SaturateSignedWordToSignedByte (SRC2[63:48]);
DEST[103:96] := SaturateSignedWordToSignedByte (SRC2[79:64]);
DEST[111:104] := SaturateSignedWordToSignedByte (SRC2[95:80]);
DEST[119:112] := SaturateSignedWordToSignedByte (SRC2[111:96]);
DEST[127:120] := SaturateSignedWordToSignedByte (SRC2[127:112]);
DEST[135:128] := SaturateSignedWordToSignedByte (SRC1[143:128]);
DEST[143:136] := SaturateSignedWordToSignedByte (SRC1[159:144]);
DEST[151:144] := SaturateSignedWordToSignedByte (SRC1[175:160]);
DEST[159:152] := SaturateSignedWordToSignedByte (SRC1[191:176]);
DEST[167:160] := SaturateSignedWordToSignedByte (SRC1[207:192]);
DEST[175:168] := SaturateSignedWordToSignedByte (SRC1[223:208]);
DEST[183:176] := SaturateSignedWordToSignedByte (SRC1[239:224]);
```

```
    DEST[191:184] := SaturateSignedWordToSignedByte (SRC1[255:240]);
    DEST[199:192] := SaturateSignedWordToSignedByte (SRC2[143:128]);
    DEST[207:200] := SaturateSignedWordToSignedByte (SRC2[159:144]);
    DEST[215:208] := SaturateSignedWordToSignedByte (SRC2[175:160]);
    DEST[223:216] := SaturateSignedWordToSignedByte (SRC2[191:176]);
    DEST[231:224] := SaturateSignedWordToSignedByte (SRC2[207:192]);
    DEST[239:232] := SaturateSignedWordToSignedByte (SRC2[223:208]);
    DEST[247:240] := SaturateSignedWordToSignedByte (SRC2[239:224]);
    DEST[255:248] := SaturateSignedWordToSignedByte (SRC2[255:240]);
    DEST[MAXVL-1:256] := 0;
```

VPACKSSDW Instruction (VEX.256 Encoded Version)

```
    DEST[15:0] := SaturateSignedDwordToSignedWord (SRC1[31:0]);
    DEST[31:16] := SaturateSignedDwordToSignedWord (SRC1[63:32]);
    DEST[47:32] := SaturateSignedDwordToSignedWord (SRC1[95:64]);
    DEST[63:48] := SaturateSignedDwordToSignedWord (SRC1[127:96]);
    DEST[79:64] := SaturateSignedDwordToSignedWord (SRC2[31:0]);
    DEST[95:80] := SaturateSignedDwordToSignedWord (SRC2[63:32]);
    DEST[111:96] := SaturateSignedDwordToSignedWord (SRC2[95:64]);
    DEST[127:112] := SaturateSignedDwordToSignedWord (SRC2[127:96]);
    DEST[143:128] := SaturateSignedDwordToSignedWord (SRC1[159:128]);
    DEST[159:144] := SaturateSignedDwordToSignedWord (SRC1[191:160]);
    DEST[175:160] := SaturateSignedDwordToSignedWord (SRC1[223:192]);
    DEST[191:176] := SaturateSignedDwordToSignedWord (SRC1[255:224]);
    DEST[207:192] := SaturateSignedDwordToSignedWord (SRC2[159:128]);
    DEST[223:208] := SaturateSignedDwordToSignedWord (SRC2[191:160]);
    DEST[239:224] := SaturateSignedDwordToSignedWord (SRC2[223:192]);
    DEST[255:240] := SaturateSignedDwordToSignedWord (SRC2[255:224]);
    DEST[MAXVL-1:256] := 0;
```

VPACKSSWB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
TMP_DEST[7:0] := SaturateSignedWordToSignedByte (SRC1[15:0]);
TMP_DEST[15:8] := SaturateSignedWordToSignedByte (SRC1[31:16]);
TMP_DEST[23:16] := SaturateSignedWordToSignedByte (SRC1[47:32]);
TMP_DEST[31:24] := SaturateSignedWordToSignedByte (SRC1[63:48]);
TMP_DEST[39:32] := SaturateSignedWordToSignedByte (SRC1[79:64]);
TMP_DEST[47:40] := SaturateSignedWordToSignedByte (SRC1[95:80]);
TMP_DEST[55:48] := SaturateSignedWordToSignedByte (SRC1[111:96]);
TMP_DEST[63:56] := SaturateSignedWordToSignedByte (SRC1[127:112]);
TMP_DEST[71:64] := SaturateSignedWordToSignedByte (SRC2[15:0]);
TMP_DEST[79:72] := SaturateSignedWordToSignedByte (SRC2[31:16]);
TMP_DEST[87:80] := SaturateSignedWordToSignedByte (SRC2[47:32]);
TMP_DEST[95:88] := SaturateSignedWordToSignedByte (SRC2[63:48]);
TMP_DEST[103:96] := SaturateSignedWordToSignedByte (SRC2[79:64]);
TMP_DEST[111:104] := SaturateSignedWordToSignedByte (SRC2[95:80]);
TMP_DEST[119:112] := SaturateSignedWordToSignedByte (SRC2[111:96]);
TMP_DEST[127:120] := SaturateSignedWordToSignedByte (SRC2[127:112]);
IF VL >= 256
    TMP_DEST[135:128] := SaturateSignedWordToSignedByte (SRC1[143:128]);
    TMP_DEST[143:136] := SaturateSignedWordToSignedByte (SRC1[159:144]);
    TMP_DEST[151:144] := SaturateSignedWordToSignedByte (SRC1[175:160]);
    TMP_DEST[159:152] := SaturateSignedWordToSignedByte (SRC1[191:176]);
    TMP_DEST[167:160] := SaturateSignedWordToSignedByte (SRC1[207:192]);
```

```
    TMP_DEST[175:168] := SaturateSignedWordToSignedByte (SRC1[223:208]);
    TMP_DEST[183:176] := SaturateSignedWordToSignedByte (SRC1[239:224]);
    TMP_DEST[191:184] := SaturateSignedWordToSignedByte (SRC1[255:240]);
    TMP_DEST[199:192] := SaturateSignedWordToSignedByte (SRC2[143:128]);
    TMP_DEST[207:200] := SaturateSignedWordToSignedByte (SRC2[159:144]);
    TMP_DEST[215:208] := SaturateSignedWordToSignedByte (SRC2[175:160]);
    TMP_DEST[223:216] := SaturateSignedWordToSignedByte (SRC2[191:176]);
    TMP_DEST[231:224] := SaturateSignedWordToSignedByte (SRC2[207:192]);
    TMP_DEST[239:232] := SaturateSignedWordToSignedByte (SRC2[223:208]);
    TMP_DEST[247:240] := SaturateSignedWordToSignedByte (SRC2[239:224]);
    TMP_DEST[255:248] := SaturateSignedWordToSignedByte (SRC2[255:240]);
FI;
IF VL >= 512
    TMP_DEST[263:256] := SaturateSignedWordToSignedByte (SRC1[271:256]);
    TMP_DEST[271:264] := SaturateSignedWordToSignedByte (SRC1[287:272]);
    TMP_DEST[279:272] := SaturateSignedWordToSignedByte (SRC1[303:288]);
    TMP_DEST[287:280] := SaturateSignedWordToSignedByte (SRC1[319:304]);
    TMP_DEST[295:288] := SaturateSignedWordToSignedByte (SRC1[335:320]);
    TMP_DEST[303:296] := SaturateSignedWordToSignedByte (SRC1[351:336]);
    TMP_DEST[311:304] := SaturateSignedWordToSignedByte (SRC1[367:352]);
    TMP_DEST[319:312] := SaturateSignedWordToSignedByte (SRC1[383:368]);

    TMP_DEST[327:320] := SaturateSignedWordToSignedByte (SRC2[271:256]);
    TMP_DEST[335:328] := SaturateSignedWordToSignedByte (SRC2[287:272]);
    TMP_DEST[343:336] := SaturateSignedWordToSignedByte (SRC2[303:288]);
    TMP_DEST[351:344] := SaturateSignedWordToSignedByte (SRC2[319:304]);
    TMP_DEST[359:352] := SaturateSignedWordToSignedByte (SRC2[335:320]);
    TMP_DEST[367:360] := SaturateSignedWordToSignedByte (SRC2[351:336]);
    TMP_DEST[375:368] := SaturateSignedWordToSignedByte (SRC2[367:352]);
    TMP_DEST[383:376] := SaturateSignedWordToSignedByte (SRC2[383:368]);

    TMP_DEST[391:384] := SaturateSignedWordToSignedByte (SRC1[399:384]);
    TMP_DEST[399:392] := SaturateSignedWordToSignedByte (SRC1[415:400]);
    TMP_DEST[407:400] := SaturateSignedWordToSignedByte (SRC1[431:416]);
    TMP_DEST[415:408] := SaturateSignedWordToSignedByte (SRC1[447:432]);
    TMP_DEST[423:416] := SaturateSignedWordToSignedByte (SRC1[463:448]);
    TMP_DEST[431:424] := SaturateSignedWordToSignedByte (SRC1[479:464]);
    TMP_DEST[439:432] := SaturateSignedWordToSignedByte (SRC1[495:480]);
    TMP_DEST[447:440] := SaturateSignedWordToSignedByte (SRC1[511:496]);

    TMP_DEST[455:448] := SaturateSignedWordToSignedByte (SRC2[399:384]);
    TMP_DEST[463:456] := SaturateSignedWordToSignedByte (SRC2[415:400]);
    TMP_DEST[471:464] := SaturateSignedWordToSignedByte (SRC2[431:416]);
    TMP_DEST[479:472] := SaturateSignedWordToSignedByte (SRC2[447:432]);
    TMP_DEST[487:480] := SaturateSignedWordToSignedByte (SRC2[463:448]);
    TMP_DEST[495:488] := SaturateSignedWordToSignedByte (SRC2[479:464]);
    TMP_DEST[503:496] := SaturateSignedWordToSignedByte (SRC2[495:480]);
    TMP_DEST[511:504] := SaturateSignedWordToSignedByte (SRC2[511:496]);
FI;
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+7:i] := TMP_DEST[i+7:i]
```

```
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPACKSSDW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO ((KL/2) - 1)
    i := j * 32

    IF (EVEX.b == 1) AND (SRC2 *is memory*)
        THEN
            TMP_SRC2[i+31:i] := SRC2[31:0]
        ELSE
            TMP_SRC2[i+31:i] := SRC2[i+31:i]
    FI;
ENDFOR;

TMP_DEST[15:0] := SaturateSignedDwordToSignedWord (SRC1[31:0]);
TMP_DEST[31:16] := SaturateSignedDwordToSignedWord (SRC1[63:32]);
TMP_DEST[47:32] := SaturateSignedDwordToSignedWord (SRC1[95:64]);
TMP_DEST[63:48] := SaturateSignedDwordToSignedWord (SRC1[127:96]);
TMP_DEST[79:64] := SaturateSignedDwordToSignedWord (TMP_SRC2[31:0]);
TMP_DEST[95:80] := SaturateSignedDwordToSignedWord (TMP_SRC2[63:32]);
TMP_DEST[111:96] := SaturateSignedDwordToSignedWord (TMP_SRC2[95:64]);
TMP_DEST[127:112] := SaturateSignedDwordToSignedWord (TMP_SRC2[127:96]);
IF VL >= 256
    TMP_DEST[143:128] := SaturateSignedDwordToSignedWord (SRC1[159:128]);
    TMP_DEST[159:144] := SaturateSignedDwordToSignedWord (SRC1[191:160]);
    TMP_DEST[175:160] := SaturateSignedDwordToSignedWord (SRC1[223:192]);
    TMP_DEST[191:176] := SaturateSignedDwordToSignedWord (SRC1[255:224]);
    TMP_DEST[207:192] := SaturateSignedDwordToSignedWord (TMP_SRC2[159:128]);
    TMP_DEST[223:208] := SaturateSignedDwordToSignedWord (TMP_SRC2[191:160]);
    TMP_DEST[239:224] := SaturateSignedDwordToSignedWord (TMP_SRC2[223:192]);
    TMP_DEST[255:240] := SaturateSignedDwordToSignedWord (TMP_SRC2[255:224]);
FI;
IF VL >= 512
    TMP_DEST[271:256] := SaturateSignedDwordToSignedWord (SRC1[287:256]);
    TMP_DEST[287:272] := SaturateSignedDwordToSignedWord (SRC1[319:288]);
    TMP_DEST[303:288] := SaturateSignedDwordToSignedWord (SRC1[351:320]);
    TMP_DEST[319:304] := SaturateSignedDwordToSignedWord (SRC1[383:352]);
    TMP_DEST[335:320] := SaturateSignedDwordToSignedWord (TMP_SRC2[287:256]);
    TMP_DEST[351:336] := SaturateSignedDwordToSignedWord (TMP_SRC2[319:288]);
    TMP_DEST[367:352] := SaturateSignedDwordToSignedWord (TMP_SRC2[351:320]);
    TMP_DEST[383:368] := SaturateSignedDwordToSignedWord (TMP_SRC2[383:352]);

    TMP_DEST[399:384] := SaturateSignedDwordToSignedWord (SRC1[415:384]);
    TMP_DEST[415:400] := SaturateSignedDwordToSignedWord (SRC1[447:416]);
    TMP_DEST[431:416] := SaturateSignedDwordToSignedWord (SRC1[479:448]);
```

```
    TMP_DEST[447:432] := SaturateSignedDwordToSignedWord (SRC1[511:480]);
    TMP_DEST[463:448] := SaturateSignedDwordToSignedWord (TMP_SRC2[415:384]);
    TMP_DEST[479:464] := SaturateSignedDwordToSignedWord (TMP_SRC2[447:416]);
    TMP_DEST[495:480] := SaturateSignedDwordToSignedWord (TMP_SRC2[479:448]);
    TMP_DEST[511:496] := SaturateSignedDwordToSignedWord (TMP_SRC2[511:480]);
FI;
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPACKSSDW__m512i _mm512_packs_epi32(__m512i m1, __m512i m2);
VPACKSSDW__m512i _mm512_mask_packs_epi32(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKSSDW__m512i _mm512_maskz_packs_epi32(__mmask32 k, __m512i m1, __m512i m2);
VPACKSSDW__m256i _mm256_mask_packs_epi32(__m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKSSDW__m256i _mm256_maskz_packs_epi32(__mmask16 k, __m256i m1, __m256i m2);
VPACKSSDW__m128i _mm_mask_packs_epi32(__m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKSSDW__m128i _mm_maskz_packs_epi32(__mmask8 k, __m128i m1, __m128i m2);
VPACKSSWB__m512i _mm512_packs_epi16(__m512i m1, __m512i m2);
VPACKSSWB__m512i _mm512_mask_packs_epi16(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKSSWB__m512i _mm512_maskz_packs_epi16(__mmask32 k, __m512i m1, __m512i m2);
VPACKSSWB__m256i _mm256_mask_packs_epi16(__m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKSSWB__m256i _mm256_maskz_packs_epi16(__mmask16 k, __m256i m1, __m256i m2);
VPACKSSWB__m128i _mm_mask_packs_epi16(__m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKSSWB__m128i _mm_maskz_packs_epi16(__mmask8 k, __m128i m1, __m128i m2);
PACKSSWB __m128i _mm_packs_epi16(__m128i m1, __m128i m2)
PACKSSDW __m128i _mm_packs_epi32(__m128i m1, __m128i m2)
VPACKSSWB __m256i _mm256_packs_epi16(__m256i m1, __m256i m2)
VPACKSSDW __m256i _mm256_packs_epi32(__m256i m1, __m256i m2)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded VPACKSSDW, see Table 2-52, “Type E4NF Class Exception Conditions.”

EVEX-encoded VPACKSSWB, see Exceptions Type E4NF.nb in Table 2-52, “Type E4NF Class Exception Conditions.”

PACKUSDW—Pack With Unsigned Saturation

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 2B /r PACKUSDW xmm1, xmm2/m128	A	V/V	SSE4_1	Convert 4 packed signed doubleword integers from xmm1 and 4 packed signed doubleword integers from xmm2/m128 into 8 packed unsigned word integers in xmm1 using unsigned saturation.
VEX.128.66.0F38 2B /r VPACKUSDW xmm1,xmm2, xmm3/m128	B	V/V	AVX	Convert 4 packed signed doubleword integers from xmm2 and 4 packed signed doubleword integers from xmm3/m128 into 8 packed unsigned word integers in xmm1 using unsigned saturation.
VEX.256.66.0F38 2B /r VPACKUSDW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Convert 8 packed signed doubleword integers from ymm2 and 8 packed signed doubleword integers from ymm3/m256 into 16 packed unsigned word integers in ymm1 using unsigned saturation.
EVEX.128.66.0F38.W0 2B /r VPACKUSDW xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Convert packed signed doubleword integers from xmm2 and packed signed doubleword integers from xmm3/m128/m32bcst into packed unsigned word integers in xmm1 using unsigned saturation under writemask k1.
EVEX.256.66.0F38.W0 2B /r VPACKUSDW ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Convert packed signed doubleword integers from ymm2 and packed signed doubleword integers from ymm3/m256/m32bcst into packed unsigned word integers in ymm1 using unsigned saturation under writemask k1.
EVEX.512.66.0F38.W0 2B /r VPACKUSDW zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512BW OR AVX10.1	Convert packed signed doubleword integers from zmm2 and packed signed doubleword integers from zmm3/m512/m32bcst into packed unsigned word integers in zmm1 using unsigned saturation under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Converts packed signed doubleword integers in the first and second source operands into packed unsigned word integers using unsigned saturation to handle overflow conditions. If the signed doubleword value is beyond the range of an unsigned word (that is, greater than FFFFH or less than 0000H), the saturated unsigned word integer value of FFFFH or 0000H, respectively, is stored in the destination.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, updated conditionally under the writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding destination register destination are unmodified.

Operation

PACKUSDW (Legacy SSE Instruction)

```
TMP[15:0] := (DEST[31:0] < 0) ? 0 : DEST[15:0];
DEST[15:0] := (DEST[31:0] > FFFFH) ? FFFFH : TMP[15:0] ;
TMP[31:16] := (DEST[63:32] < 0) ? 0 : DEST[47:32];
DEST[31:16] := (DEST[63:32] > FFFFH) ? FFFFH : TMP[31:16] ;
TMP[47:32] := (DEST[95:64] < 0) ? 0 : DEST[79:64];
DEST[47:32] := (DEST[95:64] > FFFFH) ? FFFFH : TMP[47:32] ;
TMP[63:48] := (DEST[127:96] < 0) ? 0 : DEST[111:96];
DEST[63:48] := (DEST[127:96] > FFFFH) ? FFFFH : TMP[63:48] ;
TMP[79:64] := (SRC[31:0] < 0) ? 0 : SRC[15:0];
DEST[79:64] := (SRC[31:0] > FFFFH) ? FFFFH : TMP[79:64] ;
TMP[95:80] := (SRC[63:32] < 0) ? 0 : SRC[47:32];
DEST[95:80] := (SRC[63:32] > FFFFH) ? FFFFH : TMP[95:80] ;
TMP[111:96] := (SRC[95:64] < 0) ? 0 : SRC[79:64];
DEST[111:96] := (SRC[95:64] > FFFFH) ? FFFFH : TMP[111:96] ;
TMP[127:112] := (SRC[127:96] < 0) ? 0 : SRC[111:96];
DEST[127:112] := (SRC[127:96] > FFFFH) ? FFFFH : TMP[127:112] ;
DEST[MAXVL-1:128] (Unmodified)
```

PACKUSDW (VEX.128 Encoded Version)

```
TMP[15:0] := (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
DEST[15:0] := (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0] ;
TMP[31:16] := (SRC1[63:32] < 0) ? 0 : SRC1[47:32];
DEST[31:16] := (SRC1[63:32] > FFFFH) ? FFFFH : TMP[31:16] ;
TMP[47:32] := (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
DEST[47:32] := (SRC1[95:64] > FFFFH) ? FFFFH : TMP[47:32] ;
TMP[63:48] := (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
DEST[63:48] := (SRC1[127:96] > FFFFH) ? FFFFH : TMP[63:48] ;
TMP[79:64] := (SRC2[31:0] < 0) ? 0 : SRC2[15:0];
DEST[79:64] := (SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64] ;
TMP[95:80] := (SRC2[63:32] < 0) ? 0 : SRC2[47:32];
DEST[95:80] := (SRC2[63:32] > FFFFH) ? FFFFH : TMP[95:80] ;
TMP[111:96] := (SRC2[95:64] < 0) ? 0 : SRC2[79:64];
DEST[111:96] := (SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96] ;
TMP[127:112] := (SRC2[127:96] < 0) ? 0 : SRC2[111:96];
DEST[127:112] := (SRC2[127:96] > FFFFH) ? FFFFH : TMP[127:112];
DEST[MAXVL-1:128] := 0;
```

VPACKUSDW (VEX.256 Encoded Version)

```
TMP[15:0] := (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
DEST[15:0] := (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0] ;
TMP[31:16] := (SRC1[63:32] < 0) ? 0 : SRC1[47:32];
DEST[31:16] := (SRC1[63:32] > FFFFH) ? FFFFH : TMP[31:16] ;
TMP[47:32] := (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
DEST[47:32] := (SRC1[95:64] > FFFFH) ? FFFFH : TMP[47:32] ;
TMP[63:48] := (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
DEST[63:48] := (SRC1[127:96] > FFFFH) ? FFFFH : TMP[63:48] ;
TMP[79:64] := (SRC2[31:0] < 0) ? 0 : SRC2[15:0];
DEST[79:64] := (SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64] ;
TMP[95:80] := (SRC2[63:32] < 0) ? 0 : SRC2[47:32];
DEST[95:80] := (SRC2[63:32] > FFFFH) ? FFFFH : TMP[95:80] ;
TMP[111:96] := (SRC2[95:64] < 0) ? 0 : SRC2[79:64];
DEST[111:96] := (SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96] ;
TMP[127:112] := (SRC2[127:96] < 0) ? 0 : SRC2[111:96];
DEST[127:112] := (SRC2[127:96] > FFFFH) ? FFFFH : TMP[127:112] ;
TMP[143:128] := (SRC1[159:128] < 0) ? 0 : SRC1[143:128];
DEST[143:128] := (SRC1[159:128] > FFFFH) ? FFFFH : TMP[143:128] ;
TMP[159:144] := (SRC1[191:160] < 0) ? 0 : SRC1[175:160];
DEST[159:144] := (SRC1[191:160] > FFFFH) ? FFFFH : TMP[159:144] ;
TMP[175:160] := (SRC1[223:192] < 0) ? 0 : SRC1[207:192];
DEST[175:160] := (SRC1[223:192] > FFFFH) ? FFFFH : TMP[175:160] ;
TMP[191:176] := (SRC1[255:224] < 0) ? 0 : SRC1[239:224];
DEST[191:176] := (SRC1[255:224] > FFFFH) ? FFFFH : TMP[191:176] ;
TMP[207:192] := (SRC2[159:128] < 0) ? 0 : SRC2[143:128];
DEST[207:192] := (SRC2[159:128] > FFFFH) ? FFFFH : TMP[207:192] ;
TMP[223:208] := (SRC2[191:160] < 0) ? 0 : SRC2[175:160];
DEST[223:208] := (SRC2[191:160] > FFFFH) ? FFFFH : TMP[223:208] ;
TMP[239:224] := (SRC2[223:192] < 0) ? 0 : SRC2[207:192];
DEST[239:224] := (SRC2[223:192] > FFFFH) ? FFFFH : TMP[239:224] ;
TMP[255:240] := (SRC2[255:224] < 0) ? 0 : SRC2[239:224];
DEST[255:240] := (SRC2[255:224] > FFFFH) ? FFFFH : TMP[255:240] ;
DEST[MAXVL-1:256] := 0;
```

VPACKUSDW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO ((KL/2) - 1)
    i := j * 32

    IF (EVEX.b == 1) AND (SRC2 *is memory*)
        THEN
            TMP_SRC2[i+31:i] := SRC2[31:0]
        ELSE
            TMP_SRC2[i+31:i] := SRC2[i+31:i]
    FI;
ENDFOR;

TMP[15:0] := (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
DEST[15:0] := (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0] ;
TMP[31:16] := (SRC1[63:32] < 0) ? 0 : SRC1[47:32];
DEST[31:16] := (SRC1[63:32] > FFFFH) ? FFFFH : TMP[31:16] ;
TMP[47:32] := (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
DEST[47:32] := (SRC1[95:64] > FFFFH) ? FFFFH : TMP[47:32] ;
```

```
TMP[63:48] := (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
DEST[63:48] := (SRC1[127:96] > FFFFH) ? FFFFH : TMP[63:48] ;
TMP[79:64] := (TMP_SRC2[31:0] < 0) ? 0 : TMP_SRC2[15:0];
DEST[79:64] := (TMP_SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64] ;
TMP[95:80] := (TMP_SRC2[63:32] < 0) ? 0 : TMP_SRC2[47:32];
DEST[95:80] := (TMP_SRC2[63:32] > FFFFH) ? FFFFH : TMP[95:80] ;
TMP[111:96] := (TMP_SRC2[95:64] < 0) ? 0 : TMP_SRC2[79:64];
DEST[111:96] := (TMP_SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96] ;
TMP[127:112] := (TMP_SRC2[127:96] < 0) ? 0 : TMP_SRC2[111:96];
DEST[127:112] := (TMP_SRC2[127:96] > FFFFH) ? FFFFH : TMP[127:112] ;
IF VL >= 256
    TMP[143:128] := (SRC1[159:128] < 0) ? 0 : SRC1[143:128];
    DEST[143:128] := (SRC1[159:128] > FFFFH) ? FFFFH : TMP[143:128] ;
    TMP[159:144] := (SRC1[191:160] < 0) ? 0 : SRC1[175:160];
    DEST[159:144] := (SRC1[191:160] > FFFFH) ? FFFFH : TMP[159:144] ;
    TMP[175:160] := (SRC1[223:192] < 0) ? 0 : SRC1[207:192];
    DEST[175:160] := (SRC1[223:192] > FFFFH) ? FFFFH : TMP[175:160] ;
    TMP[191:176] := (SRC1[255:224] < 0) ? 0 : SRC1[239:224];
    DEST[191:176] := (SRC1[255:224] > FFFFH) ? FFFFH : TMP[191:176] ;
    TMP[207:192] := (TMP_SRC2[159:128] < 0) ? 0 : TMP_SRC2[143:128];
    DEST[207:192] := (TMP_SRC2[159:128] > FFFFH) ? FFFFH : TMP[207:192] ;
    TMP[223:208] := (TMP_SRC2[191:160] < 0) ? 0 : TMP_SRC2[175:160];
    DEST[223:208] := (TMP_SRC2[191:160] > FFFFH) ? FFFFH : TMP[223:208] ;
    TMP[239:224] := (TMP_SRC2[223:192] < 0) ? 0 : TMP_SRC2[207:192];
    DEST[239:224] := (TMP_SRC2[223:192] > FFFFH) ? FFFFH : TMP[239:224] ;
    TMP[255:240] := (TMP_SRC2[255:224] < 0) ? 0 : TMP_SRC2[239:224];
    DEST[255:240] := (TMP_SRC2[255:224] > FFFFH) ? FFFFH : TMP[255:240] ;
FI;
IF VL >= 512
    TMP[271:256] := (SRC1[287:256] < 0) ? 0 : SRC1[271:256];
    DEST[271:256] := (SRC1[287:256] > FFFFH) ? FFFFH : TMP[271:256] ;
    TMP[287:272] := (SRC1[319:288] < 0) ? 0 : SRC1[303:288];
    DEST[287:272] := (SRC1[319:288] > FFFFH) ? FFFFH : TMP[287:272] ;
    TMP[303:288] := (SRC1[351:320] < 0) ? 0 : SRC1[335:320];
    DEST[303:288] := (SRC1[351:320] > FFFFH) ? FFFFH : TMP[303:288] ;
    TMP[319:304] := (SRC1[383:352] < 0) ? 0 : SRC1[367:352];
    DEST[319:304] := (SRC1[383:352] > FFFFH) ? FFFFH : TMP[319:304] ;
    TMP[335:320] := (TMP_SRC2[287:256] < 0) ? 0 : TMP_SRC2[271:256];
    DEST[335:304] := (TMP_SRC2[287:256] > FFFFH) ? FFFFH : TMP[79:64] ;
    TMP[351:336] := (TMP_SRC2[319:288] < 0) ? 0 : TMP_SRC2[303:288];
    DEST[351:336] := (TMP_SRC2[319:288] > FFFFH) ? FFFFH : TMP[351:336] ;
    TMP[367:352] := (TMP_SRC2[351:320] < 0) ? 0 : TMP_SRC2[315:320];
    DEST[367:352] := (TMP_SRC2[351:320] > FFFFH) ? FFFFH : TMP[367:352] ;
    TMP[383:368] := (TMP_SRC2[383:352] < 0) ? 0 : TMP_SRC2[367:352];
    DEST[383:368] := (TMP_SRC2[383:352] > FFFFH) ? FFFFH : TMP[383:368] ;
    TMP[399:384] := (SRC1[415:384] < 0) ? 0 : SRC1[399:384];
    DEST[399:384] := (SRC1[415:384] > FFFFH) ? FFFFH : TMP[399:384] ;
    TMP[415:400] := (SRC1[447:416] < 0) ? 0 : SRC1[431:416];
    DEST[415:400] := (SRC1[447:416] > FFFFH) ? FFFFH : TMP[415:400] ;
    TMP[431:416] := (SRC1[479:448] < 0) ? 0 : SRC1[463:448];
    DEST[431:416] := (SRC1[479:448] > FFFFH) ? FFFFH : TMP[431:416] ;
    TMP[447:432] := (SRC1[511:480] < 0) ? 0 : SRC1[495:480];
    DEST[447:432] := (SRC1[511:480] > FFFFH) ? FFFFH : TMP[447:432] ;
    TMP[463:448] := (TMP_SRC2[415:384] < 0) ? 0 : TMP_SRC2[399:384];
```

```
    DEST[463:448] := (TMP_SRC2[415:384] > FFFFH) ? FFFFH : TMP[463:448] ;
    TMP[475:464] := (TMP_SRC2[447:416] < 0) ? 0 : TMP_SRC2[431:416];
    DEST[475:464] := (TMP_SRC2[447:416] > FFFFH) ? FFFFH : TMP[475:464] ;
    TMP[491:476] := (TMP_SRC2[479:448] < 0) ? 0 : TMP_SRC2[463:448];
    DEST[491:476] := (TMP_SRC2[479:448] > FFFFH) ? FFFFH : TMP[491:476] ;
    TMP[511:492] := (TMP_SRC2[511:480] < 0) ? 0 : TMP_SRC2[495:480];
    DEST[511:492] := (TMP_SRC2[511:480] > FFFFH) ? FFFFH : TMP[511:492] ;
FI;
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+15:i] := TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPACKUSDW__m512i _mm512_packus_epi32(__m512i m1, __m512i m2);
VPACKUSDW__m512i _mm512_mask_packus_epi32(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKUSDW__m512i _mm512_maskz_packus_epi32(__mmask32 k, __m512i m1, __m512i m2);
VPACKUSDW__m256i _mm256_mask_packus_epi32(__m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKUSDW__m256i _mm256_maskz_packus_epi32(__mmask16 k, __m256i m1, __m256i m2);
VPACKUSDW__m128i _mm_mask_packus_epi32(__m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKUSDW__m128i _mm_maskz_packus_epi32(__mmask8 k, __m128i m1, __m128i m2);
PACKUSDW__m128i _mm_packus_epi32(__m128i m1, __m128i m2);
VPACKUSDW__m256i _mm256_packus_epi32(__m256i m1, __m256i m2);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-52, “Type E4NF Class Exception Conditions.”

PACKUSWB—Pack With Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 67 /r[1] PACKUSWB mm, mm/m64	A	V/V	MMX	Converts 4 signed word integers from mm and 4 signed word integers from mm/m64 into 8 unsigned byte integers in mm using unsigned saturation.
66 0F 67 /r PACKUSWB xmm1, xmm2/m128	A	V/V	SSE2	Converts 8 signed word integers from xmm1 and 8 signed word integers from xmm2/m128 into 16 unsigned byte integers in xmm1 using unsigned saturation.
VEX.128.66.0F.WIG 67 /r VPACKUSWB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Converts 8 signed word integers from xmm2 and 8 signed word integers from xmm3/m128 into 16 unsigned byte integers in xmm1 using unsigned saturation.
VEX.256.66.0F.WIG 67 /r VPACKUSWB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Converts 16 signed word integers from ymm2 and 16signed word integers from ymm3/m256 into 32 unsigned byte integers in ymm1 using unsigned saturation.
EVEX.128.66.0F.WIG 67 /r VPACKUSWB xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Converts signed word integers from xmm2 and signed word integers from xmm3/m128 into unsigned byte integers in xmm1 using unsigned saturation under writemask k1.
EVEX.256.66.0F.WIG 67 /r VPACKUSWB ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Converts signed word integers from ymm2 and signed word integers from ymm3/m256 into unsigned byte integers in ymm1 using unsigned saturation under writemask k1.
EVEX.512.66.0F.WIG 67 /r VPACKUSWB zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Converts signed word integers from zmm2 and signed word integers from zmm3/m512 into unsigned byte integers in zmm1 using unsigned saturation under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Converts 4, 8, 16, or 32 signed word integers from the destination operand (first operand) and 4, 8, 16, or 32 signed word integers from the source operand (second operand) into 8, 16, 32 or 64 unsigned byte integers and stores the result in the destination operand. (See Figure 4-6 for an example of the packing operation.) If a signed word integer value is beyond the range of an unsigned byte integer (that is, greater than FFH or less than 00H), the saturated unsigned byte integer value of FFH or 00H, respectively, is stored in the destination.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register or a 512-bit memory location. The destination operand is a ZMM register.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

PACKUSWB (With 64-bit Operands)

```
DEST[7:0] := SaturateSignedWordToUnsignedByte DEST[15:0];
DEST[15:8] := SaturateSignedWordToUnsignedByte DEST[31:16];
DEST[23:16] := SaturateSignedWordToUnsignedByte DEST[47:32];
DEST[31:24] := SaturateSignedWordToUnsignedByte DEST[63:48];
DEST[39:32] := SaturateSignedWordToUnsignedByte SRC[15:0];
DEST[47:40] := SaturateSignedWordToUnsignedByte SRC[31:16];
DEST[55:48] := SaturateSignedWordToUnsignedByte SRC[47:32];
DEST[63:56] := SaturateSignedWordToUnsignedByte SRC[63:48];
```

PACKUSWB (Legacy SSE Instruction)

```
DEST[7:0] := SaturateSignedWordToUnsignedByte (DEST[15:0]);
DEST[15:8] := SaturateSignedWordToUnsignedByte (DEST[31:16]);
DEST[23:16] := SaturateSignedWordToUnsignedByte (DEST[47:32]);
DEST[31:24] := SaturateSignedWordToUnsignedByte (DEST[63:48]);
DEST[39:32] := SaturateSignedWordToUnsignedByte (DEST[79:64]);
DEST[47:40] := SaturateSignedWordToUnsignedByte (DEST[95:80]);
DEST[55:48] := SaturateSignedWordToUnsignedByte (DEST[111:96]);
DEST[63:56] := SaturateSignedWordToUnsignedByte (DEST[127:112]);
DEST[71:64] := SaturateSignedWordToUnsignedByte (SRC[15:0]);
DEST[79:72] := SaturateSignedWordToUnsignedByte (SRC[31:16]);
DEST[87:80] := SaturateSignedWordToUnsignedByte (SRC[47:32]);
DEST[95:88] := SaturateSignedWordToUnsignedByte (SRC[63:48]);
DEST[103:96] := SaturateSignedWordToUnsignedByte (SRC[79:64]);
DEST[111:104] := SaturateSignedWordToUnsignedByte (SRC[95:80]);
DEST[119:112] := SaturateSignedWordToUnsignedByte (SRC[111:96]);
DEST[127:120] := SaturateSignedWordToUnsignedByte (SRC[127:112]);
```

PACKUSWB (VEX.128 Encoded Version)

```
DEST[7:0] := SaturateSignedWordToUnsignedByte (SRC1[15:0]);
DEST[15:8] := SaturateSignedWordToUnsignedByte (SRC1[31:16]);
DEST[23:16] := SaturateSignedWordToUnsignedByte (SRC1[47:32]);
DEST[31:24] := SaturateSignedWordToUnsignedByte (SRC1[63:48]);
DEST[39:32] := SaturateSignedWordToUnsignedByte (SRC1[79:64]);
DEST[47:40] := SaturateSignedWordToUnsignedByte (SRC1[95:80]);
DEST[55:48] := SaturateSignedWordToUnsignedByte (SRC1[111:96]);
DEST[63:56] := SaturateSignedWordToUnsignedByte (SRC1[127:112]);
DEST[71:64] := SaturateSignedWordToUnsignedByte (SRC2[15:0]);
DEST[79:72] := SaturateSignedWordToUnsignedByte (SRC2[31:16]);
DEST[87:80] := SaturateSignedWordToUnsignedByte (SRC2[47:32]);
DEST[95:88] := SaturateSignedWordToUnsignedByte (SRC2[63:48]);
DEST[103:96] := SaturateSignedWordToUnsignedByte (SRC2[79:64]);
DEST[111:104] := SaturateSignedWordToUnsignedByte (SRC2[95:80]);
```

DEST[119:112] := SaturateSignedWordToUnsignedByte (SRC2[111:96]);
DEST[127:120] := SaturateSignedWordToUnsignedByte (SRC2[127:112]);
DEST[MAXVL-1:128] := 0;

VPACKUSWB (VEX.256 Encoded Version)

DEST[7:0] := SaturateSignedWordToUnsignedByte (SRC1[15:0]);
DEST[15:8] := SaturateSignedWordToUnsignedByte (SRC1[31:16]);
DEST[23:16] := SaturateSignedWordToUnsignedByte (SRC1[47:32]);
DEST[31:24] := SaturateSignedWordToUnsignedByte (SRC1[63:48]);
DEST[39:32] := SaturateSignedWordToUnsignedByte (SRC1[79:64]);
DEST[47:40] := SaturateSignedWordToUnsignedByte (SRC1[95:80]);
DEST[55:48] := SaturateSignedWordToUnsignedByte (SRC1[111:96]);
DEST[63:56] := SaturateSignedWordToUnsignedByte (SRC1[127:112]);
DEST[71:64] := SaturateSignedWordToUnsignedByte (SRC2[15:0]);
DEST[79:72] := SaturateSignedWordToUnsignedByte (SRC2[31:16]);
DEST[87:80] := SaturateSignedWordToUnsignedByte (SRC2[47:32]);
DEST[95:88] := SaturateSignedWordToUnsignedByte (SRC2[63:48]);
DEST[103:96] := SaturateSignedWordToUnsignedByte (SRC2[79:64]);
DEST[111:104] := SaturateSignedWordToUnsignedByte (SRC2[95:80]);
DEST[119:112] := SaturateSignedWordToUnsignedByte (SRC2[111:96]);
DEST[127:120] := SaturateSignedWordToUnsignedByte (SRC2[127:112]);
DEST[135:128] := SaturateSignedWordToUnsignedByte (SRC1[143:128]);
DEST[143:136] := SaturateSignedWordToUnsignedByte (SRC1[159:144]);
DEST[151:144] := SaturateSignedWordToUnsignedByte (SRC1[175:160]);
DEST[159:152] := SaturateSignedWordToUnsignedByte (SRC1[191:176]);
DEST[167:160] := SaturateSignedWordToUnsignedByte (SRC1[207:192]);
DEST[175:168] := SaturateSignedWordToUnsignedByte (SRC1[223:208]);
DEST[183:176] := SaturateSignedWordToUnsignedByte (SRC1[239:224]);
DEST[191:184] := SaturateSignedWordToUnsignedByte (SRC1[255:240]);
DEST[199:192] := SaturateSignedWordToUnsignedByte (SRC2[143:128]);
DEST[207:200] := SaturateSignedWordToUnsignedByte (SRC2[159:144]);
DEST[215:208] := SaturateSignedWordToUnsignedByte (SRC2[175:160]);
DEST[223:216] := SaturateSignedWordToUnsignedByte (SRC2[191:176]);
DEST[231:224] := SaturateSignedWordToUnsignedByte (SRC2[207:192]);
DEST[239:232] := SaturateSignedWordToUnsignedByte (SRC2[223:208]);
DEST[247:240] := SaturateSignedWordToUnsignedByte (SRC2[239:224]);
DEST[255:248] := SaturateSignedWordToUnsignedByte (SRC2[255:240]);

VPACKUSWB (EVEX Encoded Versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)
TMP_DEST[7:0] := SaturateSignedWordToUnsignedByte (SRC1[15:0]);
TMP_DEST[15:8] := SaturateSignedWordToUnsignedByte (SRC1[31:16]);
TMP_DEST[23:16] := SaturateSignedWordToUnsignedByte (SRC1[47:32]);
TMP_DEST[31:24] := SaturateSignedWordToUnsignedByte (SRC1[63:48]);
TMP_DEST[39:32] := SaturateSignedWordToUnsignedByte (SRC1[79:64]);
TMP_DEST[47:40] := SaturateSignedWordToUnsignedByte (SRC1[95:80]);
TMP_DEST[55:48] := SaturateSignedWordToUnsignedByte (SRC1[111:96]);
TMP_DEST[63:56] := SaturateSignedWordToUnsignedByte (SRC1[127:112]);
TMP_DEST[71:64] := SaturateSignedWordToUnsignedByte (SRC2[15:0]);
TMP_DEST[79:72] := SaturateSignedWordToUnsignedByte (SRC2[31:16]);
TMP_DEST[87:80] := SaturateSignedWordToUnsignedByte (SRC2[47:32]);
TMP_DEST[95:88] := SaturateSignedWordToUnsignedByte (SRC2[63:48]);
TMP_DEST[103:96] := SaturateSignedWordToUnsignedByte (SRC2[79:64]);
TMP_DEST[111:104] := SaturateSignedWordToUnsignedByte (SRC2[95:80]);

```
TMP_DEST[119:112] := SaturateSignedWordToUnsignedByte (SRC2[111:96]);
TMP_DEST[127:120] := SaturateSignedWordToUnsignedByte (SRC2[127:112]);
IF VL >= 256
    TMP_DEST[135:128] := SaturateSignedWordToUnsignedByte (SRC1[143:128]);
    TMP_DEST[143:136] := SaturateSignedWordToUnsignedByte (SRC1[159:144]);
    TMP_DEST[151:144] := SaturateSignedWordToUnsignedByte (SRC1[175:160]);
    TMP_DEST[159:152] := SaturateSignedWordToUnsignedByte (SRC1[191:176]);
    TMP_DEST[167:160] := SaturateSignedWordToUnsignedByte (SRC1[207:192]);
    TMP_DEST[175:168] := SaturateSignedWordToUnsignedByte (SRC1[223:208]);
    TMP_DEST[183:176] := SaturateSignedWordToUnsignedByte (SRC1[239:224]);
    TMP_DEST[191:184] := SaturateSignedWordToUnsignedByte (SRC1[255:240]);
    TMP_DEST[199:192] := SaturateSignedWordToUnsignedByte (SRC2[143:128]);
    TMP_DEST[207:200] := SaturateSignedWordToUnsignedByte (SRC2[159:144]);
    TMP_DEST[215:208] := SaturateSignedWordToUnsignedByte (SRC2[175:160]);
    TMP_DEST[223:216] := SaturateSignedWordToUnsignedByte (SRC2[191:176]);
    TMP_DEST[231:224] := SaturateSignedWordToUnsignedByte (SRC2[207:192]);
    TMP_DEST[239:232] := SaturateSignedWordToUnsignedByte (SRC2[223:208]);
    TMP_DEST[247:240] := SaturateSignedWordToUnsignedByte (SRC2[239:224]);
    TMP_DEST[255:248] := SaturateSignedWordToUnsignedByte (SRC2[255:240]);
FI;
IF VL >= 512
    TMP_DEST[263:256] := SaturateSignedWordToUnsignedByte (SRC1[271:256]);
    TMP_DEST[271:264] := SaturateSignedWordToUnsignedByte (SRC1[287:272]);
    TMP_DEST[279:272] := SaturateSignedWordToUnsignedByte (SRC1[303:288]);
    TMP_DEST[287:280] := SaturateSignedWordToUnsignedByte (SRC1[319:304]);
    TMP_DEST[295:288] := SaturateSignedWordToUnsignedByte (SRC1[335:320]);
    TMP_DEST[303:296] := SaturateSignedWordToUnsignedByte (SRC1[351:336]);
    TMP_DEST[311:304] := SaturateSignedWordToUnsignedByte (SRC1[367:352]);
    TMP_DEST[319:312] := SaturateSignedWordToUnsignedByte (SRC1[383:368]);

    TMP_DEST[327:320] := SaturateSignedWordToUnsignedByte (SRC2[271:256]);
    TMP_DEST[335:328] := SaturateSignedWordToUnsignedByte (SRC2[287:272]);
    TMP_DEST[343:336] := SaturateSignedWordToUnsignedByte (SRC2[303:288]);
    TMP_DEST[351:344] := SaturateSignedWordToUnsignedByte (SRC2[319:304]);
    TMP_DEST[359:352] := SaturateSignedWordToUnsignedByte (SRC2[335:320]);
    TMP_DEST[367:360] := SaturateSignedWordToUnsignedByte (SRC2[351:336]);
    TMP_DEST[375:368] := SaturateSignedWordToUnsignedByte (SRC2[367:352]);
    TMP_DEST[383:376] := SaturateSignedWordToUnsignedByte (SRC2[383:368]);

    TMP_DEST[391:384] := SaturateSignedWordToUnsignedByte (SRC1[399:384]);
    TMP_DEST[399:392] := SaturateSignedWordToUnsignedByte (SRC1[415:400]);
    TMP_DEST[407:400] := SaturateSignedWordToUnsignedByte (SRC1[431:416]);
    TMP_DEST[415:408] := SaturateSignedWordToUnsignedByte (SRC1[447:432]);
    TMP_DEST[423:416] := SaturateSignedWordToUnsignedByte (SRC1[463:448]);
    TMP_DEST[431:424] := SaturateSignedWordToUnsignedByte (SRC1[479:464]);
    TMP_DEST[439:432] := SaturateSignedWordToUnsignedByte (SRC1[495:480]);
    TMP_DEST[447:440] := SaturateSignedWordToUnsignedByte (SRC1[511:496]);

    TMP_DEST[455:448] := SaturateSignedWordToUnsignedByte (SRC2[399:384]);
    TMP_DEST[463:456] := SaturateSignedWordToUnsignedByte (SRC2[415:400]);
    TMP_DEST[471:464] := SaturateSignedWordToUnsignedByte (SRC2[431:416]);
    TMP_DEST[479:472] := SaturateSignedWordToUnsignedByte (SRC2[447:432]);
    TMP_DEST[487:480] := SaturateSignedWordToUnsignedByte (SRC2[463:448]);
    TMP_DEST[495:488] := SaturateSignedWordToUnsignedByte (SRC2[479:464]);
```

```
    TMP_DEST[503:496] := SaturateSignedWordToUnsignedByte (SRC2[495:480]);
    TMP_DEST[511:504] := SaturateSignedWordToUnsignedByte (SRC2[511:496]);
FI;
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+7:i] := TMP_DEST[i+7:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPACKUSWB __m512i _mm512_packus_epi16(__m512i m1, __m512i m2);
VPACKUSWB __m512i _mm512_mask_packus_epi16(__m512i s, __mmask64 k, __m512i m1, __m512i m2);
VPACKUSWB __m512i _mm512_maskz_packus_epi16(__mmask64 k, __m512i m1, __m512i m2);
VPACKUSWB __m256i _mm256_mask_packus_epi16(__m256i s, __mmask32 k, __m256i m1, __m256i m2);
VPACKUSWB __m256i _mm256_maskz_packus_epi16(__mmask32 k, __m256i m1, __m256i m2);
VPACKUSWB __m128i _mm_mask_packus_epi16(__m128i s, __mmask16 k, __m128i m1, __m128i m2);
VPACKUSWB __m128i _mm_maskz_packus_epi16(__mmask16 k, __m128i m1, __m128i m2);
PACKUSWB __m64 _mm_packs_pu16(__m64 m1, __m64 m2)
(V)PACKUSWB __m128i _mm_packus_epi16(__m128i m1, __m128i m2)
VPACKUSWB __m256i _mm256_packus_epi16(__m256i m1, __m256i m2);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."

EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."

PADDB/PADDW/PADDD/PADDQ—Add Packed Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F FC /r[1] PADDB mm, mm/m64	A	V/V	MMX	Add packed byte integers from mm/m64 and mm.
NP 0F FD /r[1] PADDW mm, mm/m64	A	V/V	MMX	Add packed word integers from mm/m64 and mm.
NP 0F FE /r[1] PADDD mm, mm/m64	A	V/V	MMX	Add packed doubleword integers from mm/m64 and mm.
NP 0F D4 /r[1] PADDQ mm, mm/m64	A	V/V	SSE2	Add packed quadword integers from mm/m64 and mm.
66 0F FC /r PADDB xmm1, xmm2/m128	A	V/V	SSE2	Add packed byte integers from xmm2/m128 and xmm1.
66 0F FD /r PADDW xmm1, xmm2/m128	A	V/V	SSE2	Add packed word integers from xmm2/m128 and xmm1.
66 0F FE /r PADDD xmm1, xmm2/m128	A	V/V	SSE2	Add packed doubleword integers from xmm2/m128 and xmm1.
66 0F D4 /r PADDQ xmm1, xmm2/m128	A	V/V	SSE2	Add packed quadword integers from xmm2/m128 and xmm1.
VEX.128.66.0F.WIG FC /r VPADDB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed byte integers from xmm2, and xmm3/m128 and store in xmm1.
VEX.128.66.0F.WIG FD /r VPADDW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed word integers from xmm2, xmm3/m128 and store in xmm1.
VEX.128.66.0F.WIG FE /r VPADDD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed doubleword integers from xmm2, xmm3/m128 and store in xmm1.
VEX.128.66.0F.WIG D4 /r VPADDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed quadword integers from xmm2, xmm3/m128 and store in xmm1.
VEX.256.66.0F.WIG FC /r VPADDB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add packed byte integers from ymm2, and ymm3/m256 and store in ymm1.
VEX.256.66.0F.WIG FD /r VPADDW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add packed word integers from ymm2, ymm3/m256 and store in ymm1.
VEX.256.66.0F.WIG FE /r VPADDD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add packed doubleword integers from ymm2, ymm3/m256 and store in ymm1.
VEX.256.66.0F.WIG D4 /r VPADDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add packed quadword integers from ymm2, ymm3/m256 and store in ymm1.
EVEX.128.66.0F.WIG FC /r VPADDB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Add packed byte integers from xmm2, and xmm3/m128 and store in xmm1 using writemask k1.
EVEX.128.66.0F.WIG FD /r VPADDW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Add packed word integers from xmm2, and xmm3/m128 and store in xmm1 using writemask k1.
EVEX.128.66.0F.W0 FE /r VPADDD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Add packed doubleword integers from xmm2, and xmm3/m128/m32bcst and store in xmm1 using writemask k1.
EVEX.128.66.0F.W1 D4 /r VPADDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Add packed quadword integers from xmm2, and xmm3/m128/m64bcst and store in xmm1 using writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F.WIG FC /r VPADDB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Add packed byte integers from ymm2, and ymm3/m256 and store in ymm1 using writemask k1.
EVEX.256.66.0F.WIG FD /r VPADDW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Add packed word integers from ymm2, and ymm3/m256 and store in ymm1 using writemask k1.
EVEX.256.66.0F.W0 FE /r VPADDD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Add packed doubleword integers from ymm2, ymm3/m256/m32bcst and store in ymm1 using writemask k1.
EVEX.256.66.0F.W1 D4 /r VPADDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Add packed quadword integers from ymm2, ymm3/m256/m64bcst and store in ymm1 using writemask k1.
EVEX.512.66.0F.WIG FC /r VPADDB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Add packed byte integers from zmm2, and zmm3/m512 and store in zmm1 using writemask k1.
EVEX.512.66.0F.WIG FD /r VPADDW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Add packed word integers from zmm2, and zmm3/m512 and store in zmm1 using writemask k1.
EVEX.512.66.0F.W0 FE /r VPADDD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F OR AVX10.1	Add packed doubleword integers from zmm2, zmm3/m512/m32bcst and store in zmm1 using writemask k1.
EVEX.512.66.0F.W1 D4 /r VPADDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	D	V/V	AVX512F OR AVX10.1	Add packed quadword integers from zmm2, zmm3/m512/m64bcst and store in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD add of the packed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

The PADDB and VPADDB instructions add packed byte integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 8 bits (overflow), the result is wrapped around and the low 8 bits are written to the destination operand (that is, the carry is ignored).

The PADDW and VPADDW instructions add packed word integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to

be represented in 16 bits (overflow), the result is wrapped around and the low 16 bits are written to the destination operand (that is, the carry is ignored).

The PADDD and VPADDD instructions add packed doubleword integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 32 bits (overflow), the result is wrapped around and the low 32 bits are written to the destination operand (that is, the carry is ignored).

The PADDQ and VPADDQ instructions add packed quadword integers from the first source operand and second source operand and store the packed integer results in the destination operand. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination operand (that is, the carry is ignored).

Note that the (V)PADDB, (V)PADDW, (V)PADDD and (V)PADDQ instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values operated on.

EVEX encoded VPADDD/Q: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

EVEX encoded VPADDB/W: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. the upper bits (MAXVL-1:256) of the destination are cleared.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

PADDB (With 64-bit Operands)

```
DEST[7:0] := DEST[7:0] + SRC[7:0];
(* Repeat add operation for 2nd through 7th byte *)
DEST[63:56] := DEST[63:56] + SRC[63:56];
```

PADDW (With 64-bit Operands)

```
DEST[15:0] := DEST[15:0] + SRC[15:0];
(* Repeat add operation for 2nd and 3th word *)
DEST[63:48] := DEST[63:48] + SRC[63:48];
```

PADDD (With 64-bit Operands)

```
DEST[31:0] := DEST[31:0] + SRC[31:0];
DEST[63:32] := DEST[63:32] + SRC[63:32];
```

PADDQ (With 64-Bit Operands)

```
DEST[63:0] := DEST[63:0] + SRC[63:0];
```

PADDB (Legacy SSE Instruction)

```
DEST[7:0] := DEST[7:0] + SRC[7:0];
(* Repeat add operation for 2nd through 15th byte *)
DEST[127:120] := DEST[127:120] + SRC[127:120];
DEST[MAXVL-1:128] (Unmodified)
```

PADDW (Legacy SSE Instruction)

```
DEST[15:0] := DEST[15:0] + SRC[15:0];
(* Repeat add operation for 2nd through 7th word *)
DEST[127:112] := DEST[127:112] + SRC[127:112];
DEST[MAXVL-1:128] (Unmodified)
```

PADDD (Legacy SSE Instruction)

```
DEST[31:0] := DEST[31:0] + SRC[31:0];
(* Repeat add operation for 2nd and 3th doubleword *)
DEST[127:96] := DEST[127:96] + SRC[127:96];
DEST[MAXVL-1:128] (Unmodified)
```

PADDQ (Legacy SSE Instruction)

```
DEST[63:0] := DEST[63:0] + SRC[63:0];
DEST[127:64] := DEST[127:64] + SRC[127:64];
DEST[MAXVL-1:128] (Unmodified)
```

VPADDB (VEX.128 Encoded Instruction)

```
DEST[7:0] := SRC1[7:0] + SRC2[7:0];
(* Repeat add operation for 2nd through 15th byte *)
DEST[127:120] := SRC1[127:120] + SRC2[127:120];
DEST[MAXVL-1:128] := 0;
```

VPADDW (VEX.128 Encoded Instruction)

```
DEST[15:0] := SRC1[15:0] + SRC2[15:0];
(* Repeat add operation for 2nd through 7th word *)
DEST[127:112] := SRC1[127:112] + SRC2[127:112];
DEST[MAXVL-1:128] := 0;
```

VPADDD (VEX.128 Encoded Instruction)

```
DEST[31:0] := SRC1[31:0] + SRC2[31:0];
(* Repeat add operation for 2nd and 3th doubleword *)
DEST[127:96] := SRC1[127:96] + SRC2[127:96];
DEST[MAXVL-1:128] := 0;
```

VPADDQ (VEX.128 Encoded Instruction)

```
DEST[63:0] := SRC1[63:0] + SRC2[63:0];
DEST[127:64] := SRC1[127:64] + SRC2[127:64];
DEST[MAXVL-1:128] := 0;
```

VPADDB (VEX.256 Encoded Instruction)

```
DEST[7:0] := SRC1[7:0] + SRC2[7:0];
(* Repeat add operation for 2nd through 31th byte *)
DEST[255:248] := SRC1[255:248] + SRC2[255:248];
```

VPADDW (VEX.256 Encoded Instruction)

```
DEST[15:0] := SRC1[15:0] + SRC2[15:0];
(* Repeat add operation for 2nd through 15th word *)
DEST[255:240] := SRC1[255:240] + SRC2[255:240];
```

VPADDD (VEX.256 Encoded Instruction)

```
DEST[31:0] := SRC1[31:0] + SRC2[31:0];
(* Repeat add operation for 2nd and 7th doubleword *)
DEST[255:224] := SRC1[255:224] + SRC2[255:224];
```

VPADDQ (VEX.256 Encoded Instruction)

```
DEST[63:0] := SRC1[63:0] + SRC2[63:0];
DEST[127:64] := SRC1[127:64] + SRC2[127:64];
DEST[191:128] := SRC1[191:128] + SRC2[191:128];
DEST[255:192] := SRC1[255:192] + SRC2[255:192];
```

VPADDB (EVEX Encoded Versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

```
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SRC1[i+7:i] + SRC2[i+7:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPADDW (EVEX Encoded Versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)

```
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SRC1[i+15:i] + SRC2[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPADDD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := SRC1[i+31:i] + SRC2[31:0]
                ELSE DEST[i+31:i] := SRC1[i+31:i] + SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPADDQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := SRC1[i+63:i] + SRC2[63:0]
                ELSE DEST[i+63:i] := SRC1[i+63:i] + SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPADDB__m512i _mm512_add_epi8 (__m512i a, __m512i b)
VPADDW__m512i _mm512_add_epi16 (__m512i a, __m512i b)
VPADDB__m512i _mm512_mask_add_epi8 (__m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDW__m512i _mm512_mask_add_epi16 (__m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDB__m512i _mm512_maskz_add_epi8 (__mmask64 m, __m512i a, __m512i b)
VPADDW__m512i _mm512_maskz_add_epi16 (__mmask32 m, __m512i a, __m512i b)
VPADDB__m256i _mm256_mask_add_epi8 (__m256i s, __mmask32 m, __m256i a, __m256i b)
VPADDW__m256i _mm256_mask_add_epi16 (__m256i s, __mmask16 m, __m256i a, __m256i b)
VPADDB__m256i _mm256_maskz_add_epi8 (__mmask32 m, __m256i a, __m256i b)
VPADDW__m256i _mm256_maskz_add_epi16 (__mmask16 m, __m256i a, __m256i b)
VPADDB__m128i _mm_mask_add_epi8 (__m128i s, __mmask16 m, __m128i a, __m128i b)
VPADDW__m128i _mm_mask_add_epi16 (__m128i s, __mmask8 m, __m128i a, __m128i b)

VPADDB__m128i _mm_maskz_add_epi8 (__mmask16 m, __m128i a, __m128i b)
VPADDW__m128i _mm_maskz_add_epi16 (__mmask8 m, __m128i a, __m128i b)
VPADDD __m512i _mm512_add_epi32(__m512i a, __m512i b);
VPADDD __m512i _mm512_mask_add_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPADDD __m512i _mm512_maskz_add_epi32(__mmask16 k, __m512i a, __m512i b);
VPADDD __m256i _mm256_mask_add_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPADDD __m256i _mm256_maskz_add_epi32(__mmask8 k, __m256i a, __m256i b);
VPADDD __m128i _mm_mask_add_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPADDD __m128i _mm_maskz_add_epi32(__mmask8 k, __m128i a, __m128i b);
VPADDQ __m512i _mm512_add_epi64(__m512i a, __m512i b);
VPADDQ __m512i _mm512_mask_add_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPADDQ __m512i _mm512_maskz_add_epi64(__mmask8 k, __m512i a, __m512i b);
VPADDQ __m256i _mm256_mask_add_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPADDQ __m256i _mm256_maskz_add_epi64(__mmask8 k, __m256i a, __m256i b);
VPADDQ __m128i _mm_mask_add_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPADDQ __m128i _mm_maskz_add_epi64(__mmask8 k, __m128i a, __m128i b);
PADDB __m128i _mm_add_epi8 (__m128i a,__m128i b);
PADDW __m128i _mm_add_epi16 (__m128i a, __m128i b);
PADDD __m128i _mm_add_epi32 (__m128i a, __m128i b);
PADDQ __m128i _mm_add_epi64 (__m128i a, __m128i b);
VPADDB __m256i _mm256_add_epi8 (__m256ia,__m256i b);
VPADDW __m256i _mm256_add_epi16 (__m256i a, __m256i b);
VPADDD __m256i _mm256_add_epi32 (__m256i a, __m256i b);
VPADDQ __m256i _mm256_add_epi64 (__m256i a, __m256i b);
PADDB __m64 _mm_add_pi8(__m64 m1, __m64 m2)
PADDW __m64 _mm_add_pi16(__m64 m1, __m64 m2)
PADDD __m64 _mm_add_pi32(__m64 m1, __m64 m2)
PADDQ __m64 _mm_add_si64(__m64 m1, __m64 m2)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."

EVEX-encoded VPADDD/Q, see Table 2-51, "Type E4 Class Exception Conditions."

EVEX-encoded VPADDB/W, see Exceptions Type E4.nb in Table 2-51, "Type E4 Class Exception Conditions."

PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EC /r[1] PADDSB mm, mm/m64	A	V/V	MMX	Add packed signed byte integers from mm/m64 and mm and saturate the results.
66 0F EC /r PADDSB xmm1, xmm2/m128	A	V/V	SSE2	Add packed signed byte integers from xmm2/m128 and xmm1 saturate the results.
NP 0F ED /r[1] PADDSW mm, mm/m64	A	V/V	MMX	Add packed signed word integers from mm/m64 and mm and saturate the results.
66 0F ED /r PADDSW xmm1, xmm2/m128	A	V/V	SSE2	Add packed signed word integers from xmm2/m128 and xmm1 and saturate the results.
VEX.128.66.0F.WIG EC /r VPADDSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed signed byte integers from xmm3/m128 and xmm2 saturate the results.
VEX.128.66.0F.WIG ED /r VPADDSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed signed word integers from xmm3/m128 and xmm2 and saturate the results.
VEX.256.66.0F.WIG EC /r VPADDSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add packed signed byte integers from ymm2, and ymm3/m256 and store the saturated results in ymm1.
VEX.256.66.0F.WIG ED /r VPADDSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add packed signed word integers from ymm2, and ymm3/m256 and store the saturated results in ymm1.
EVEX.128.66.0F.WIG EC /r VPADDSB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Add packed signed byte integers from xmm2, and xmm3/m128 and store the saturated results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG EC /r VPADDSB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Add packed signed byte integers from ymm2, and ymm3/m256 and store the saturated results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG EC /r VPADDSB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Add packed signed byte integers from zmm2, and zmm3/m512 and store the saturated results in zmm1 under writemask k1.
EVEX.128.66.0F.WIG ED /r VPADDSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Add packed signed word integers from xmm2, and xmm3/m128 and store the saturated results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG ED /r VPADDSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Add packed signed word integers from ymm2, and ymm3/m256 and store the saturated results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG ED /r VPADDSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Add packed signed word integers from zmm2, and zmm3/m512 and store the saturated results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD add of the packed signed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

(V)PADDSB performs a SIMD add of the packed signed integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

(V)PADDSW performs a SIMD add of the packed signed word integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

EVEX encoded versions: The first source operand is an ZMM/YMM/XMM register. The second source operand is an ZMM/YMM/XMM register or a memory location. The destination operand is an ZMM/YMM/XMM register.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

PADDSB (With 64-bit Operands)

```
DEST[7:0] := SaturateToSignedByte(DEST[7:0] + SRC (7:0]);
(* Repeat add operation for 2nd through 7th bytes *)
DEST[63:56] := SaturateToSignedByte(DEST[63:56] + SRC[63:56] );
```

PADDSB (With 128-bit Operands)

```
DEST[7:0] := SaturateToSignedByte (DEST[7:0] + SRC[7:0]);
(* Repeat add operation for 2nd through 14th bytes *)
DEST[127:120] := SaturateToSignedByte (DEST[111:120] + SRC[127:120]);
```

VPADDSB (VEX.128 Encoded Version)

```
DEST[7:0] := SaturateToSignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] := SaturateToSignedByte (SRC1[111:120] + SRC2[127:120]);
DEST[MAXVL-1:128] := 0
```

VPADDSB (VEX.256 Encoded Version)

```
DEST[7:0] := SaturateToSignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat add operation for 2nd through 31st bytes *)
DEST[255:248] := SaturateToSignedByte (SRC1[255:248] + SRC2[255:248]);
```

VPADDSB (EVEX Encoded Versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)

```
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SaturateToSignedByte (SRC1[i+7:i] + SRC2[i+7:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

PADDSW (with 64-bit operands)

```
    DEST[15:0] := SaturateToSignedWord(DEST[15:0] + SRC[15:0] );
    (* Repeat add operation for 2nd and 7th words *)
    DEST[63:48] := SaturateToSignedWord(DEST[63:48] + SRC[63:48] );
```

PADDSW (with 128-bit operands)

```
    DEST[15:0] := SaturateToSignedWord (DEST[15:0] + SRC[15:0]);
    (* Repeat add operation for 2nd through 7th words *)
    DEST[127:112] := SaturateToSignedWord (DEST[127:112] + SRC[127:112]);
```

VPADDSW (VEX.128 Encoded Version)

```
    DEST[15:0] := SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
    (* Repeat subtract operation for 2nd through 7th words *)
    DEST[127:112] := SaturateToSignedWord (SRC1[127:112] + SRC2[127:112]);
    DEST[MAXVL-1:128] := 0
```

VPADDSW (VEX.256 Encoded Version)

```
    DEST[15:0] := SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
    (* Repeat add operation for 2nd through 15th words *)
    DEST[255:240] := SaturateToSignedWord (SRC1[255:240] + SRC2[255:240])
```

VPADDSW (EVEX Encoded Versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)

```
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SaturateToSignedWord (SRC1[i+15:i] + SRC2[i+15:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PADDSB __m64 _mm_adds_pi8(__m64 m1, __m64 m2)
(V)PADDSB __m128i _mm_adds_epi8 (__m128i a, __m128i b)
VPADDSB __m256i _mm256_adds_epi8 (__m256i a, __m256i b)
PADDSW __m64 _mm_adds_pi16(__m64 m1, __m64 m2)
(V)PADDSW __m128i _mm_adds_epi16 (__m128i a, __m128i b)
VPADDSW __m256i _mm256_adds_epi16 (__m256i a, __m256i b)
VPADDSB __m512i _mm512_adds_epi8 (__m512i a, __m512i b)
VPADDSW __m512i _mm512_adds_epi16 (__m512i a, __m512i b)
VPADDSB __m512i _mm512_mask_adds_epi8 (__m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDSW __m512i _mm512_mask_adds_epi16 (__m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDSB __m512i _mm512_maskz_adds_epi8 (__mmask64 m, __m512i a, __m512i b)
VPADDSW __m512i _mm512_maskz_adds_epi16 (__mmask32 m, __m512i a, __m512i b)
VPADDSB __m256i _mm256_mask_adds_epi8 (__m256i s, __mmask32 m, __m256i a, __m256i b)
VPADDSW __m256i _mm256_mask_adds_epi16 (__m256i s, __mmask16 m, __m256i a, __m256i b)
VPADDSB __m256i _mm256_maskz_adds_epi8 (__mmask32 m, __m256i a, __m256i b)
VPADDSW __m256i _mm256_maskz_adds_epi16 (__mmask16 m, __m256i a, __m256i b)
VPADDSB __m128i _mm_mask_adds_epi8 (__m128i s, __mmask16 m, __m128i a, __m128i b)
VPADDSW __m128i _mm_mask_adds_epi16 (__m128i s, __mmask8 m, __m128i a, __m128i b)
VPADDSB __m128i _mm_maskz_adds_epi8 (__mmask16 m, __m128i a, __m128i b)
VPADDSW __m128i _mm_maskz_adds_epi16 (__mmask8 m, __m128i a, __m128i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

PADDUSB/PADDUSW—Add Packed Unsigned Integers With Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DC /r[1] PADDUSB mm, mm/m64	A	V/V	MMX	Add packed unsigned byte integers from mm/m64 and mm and saturate the results.
66 0F DC /r PADDUSB xmm1, xmm2/m128	A	V/V	SSE2	Add packed unsigned byte integers from xmm2/m128 and xmm1 saturate the results.
NP 0F DD /r[1] PADDUSW mm, mm/m64	A	V/V	MMX	Add packed unsigned word integers from mm/m64 and mm and saturate the results.
66 0F DD /r PADDUSW xmm1, xmm2/m128	A	V/V	SSE2	Add packed unsigned word integers from xmm2/m128 to xmm1 and saturate the results.
VEX.128.660F.WIG DC /r VPADDUSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed unsigned byte integers from xmm3/m128 to xmm2 and saturate the results.
VEX.128.66.0F.WIG DD /r VPADDUSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed unsigned word integers from xmm3/m128 to xmm2 and saturate the results.
VEX.256.66.0F.WIG DC /r VPADDUSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add packed unsigned byte integers from ymm2, and ymm3/m256 and store the saturated results in ymm1.
VEX.256.66.0F.WIG DD /r VPADDUSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add packed unsigned word integers from ymm2, and ymm3/m256 and store the saturated results in ymm1.
EVEX.128.66.0F.WIG DC /r VPADDUSB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Add packed unsigned byte integers from xmm2, and xmm3/m128 and store the saturated results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG DC /r VPADDUSB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Add packed unsigned byte integers from ymm2, and ymm3/m256 and store the saturated results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG DC /r VPADDUSB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Add packed unsigned byte integers from zmm2, and zmm3/m512 and store the saturated results in zmm1 under writemask k1.
EVEX.128.66.0F.WIG DD /r VPADDUSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Add packed unsigned word integers from xmm2, and xmm3/m128 and store the saturated results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG DD /r VPADDUSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Add packed unsigned word integers from ymm2, and ymm3/m256 and store the saturated results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG DD /r VPADDUSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Add packed unsigned word integers from zmm2, and zmm3/m512 and store the saturated results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD add of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

(V)PADDUSB performs a SIMD add of the packed unsigned integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual byte result is beyond the range of an unsigned byte integer (that is, greater than FFH), the saturated value of FFH is written to the destination operand.

(V)PADDUSW performs a SIMD add of the packed unsigned word integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual word result is beyond the range of an unsigned word integer (that is, greater than FFFFH), the saturated value of FFFFH is written to the destination operand.

EVEX encoded versions: The first source operand is an ZMM/YMM/XMM register. The second source operand is an ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination is an ZMM/YMM/XMM register.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding destination register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

PADDUSB (With 64-bit Operands)

```
DEST[7:0] := SaturateToUnsignedByte(DEST[7:0] + SRC (7:0] );
(* Repeat add operation for 2nd through 7th bytes *)
DEST[63:56] := SaturateToUnsignedByte(DEST[63:56] + SRC[63:56]
```

PADDUSB (With 128-bit Operands)

```
DEST[7:0] := SaturateToUnsignedByte (DEST[7:0] + SRC[7:0]);
(* Repeat add operation for 2nd through 14th bytes *)
DEST[127:120] := SaturateToUnSignedByte (DEST[127:120] + SRC[127:120]);
```

VPADDUSB (VEX.128 Encoded Version)

```
DEST[7:0] := SaturateToUnsignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] := SaturateToUnsignedByte (SRC1[111:120] + SRC2[127:120]);
DEST[MAXVL-1:128] := 0
```

VPADDUSB (VEX.256 Encoded Version)

```
DEST[7:0] := SaturateToUnsignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat add operation for 2nd through 31st bytes *)
DEST[255:248] := SaturateToUnsignedByte (SRC1[255:248] + SRC2[255:248]);
```

PADDUSW (With 64-bit Operands)

```
DEST[15:0] := SaturateToUnsignedWord(DEST[15:0] + SRC[15:0] );
(* Repeat add operation for 2nd and 3rd words *)
DEST[63:48] := SaturateToUnsignedWord(DEST[63:48] + SRC[63:48] );
```

PADDUSW (With 128-bit Operands)

```
DEST[15:0] := SaturateToUnsignedWord (DEST[15:0] + SRC[15:0]);
(* Repeat add operation for 2nd through 7th words *)
DEST[127:112] := SaturateToUnSignedWord (DEST[127:112] + SRC[127:112]);
```

VPADDUSW (VEX.128 Encoded Version)

```
DEST[15:0] := SaturateToUnsignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] := SaturateToUnsignedWord (SRC1[127:112] + SRC2[127:112]);
DEST[MAXVL-1:128] := 0
```

VPADDUSW (VEX.256 Encoded Version)

```
DEST[15:0] := SaturateToUnsignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat add operation for 2nd through 15th words *)
DEST[255:240] := SaturateToUnsignedWord (SRC1[255:240] + SRC2[255:240])
```

VPADDUSB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SaturateToUnsignedByte (SRC1[i+7:i] + SRC2[i+7:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPADDUSW (EVEX Encoded Versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)

```
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SaturateToUnsignedWord (SRC1[i+15:i] + SRC2[i+15:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PADDUSB __m64 _mm_adds_pu8(__m64 m1, __m64 m2)
PADDUSW __m64 _mm_adds_pu16(__m64 m1, __m64 m2)
(V)PADDUSB __m128i _mm_adds_epu8 (__m128i a, __m128i b)
(V)PADDUSW __m128i _mm_adds_epu16 (__m128i a, __m128i b)
VPADDUSB __m256i _mm256_adds_epu8 (__m256i a, __m256i b)
VPADDUSW __m256i _mm256_adds_epu16 (__m256i a, __m256i b)
VPADDUSB __m512i _mm512_adds_epu8 (__m512i a, __m512i b)
VPADDUSW __m512i _mm512_adds_epu16 (__m512i a, __m512i b)
VPADDUSB __m512i _mm512_mask_adds_epu8 (__m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDUSW __m512i _mm512_mask_adds_epu16 (__m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDUSB __m512i _mm512_maskz_adds_epu8 (__mmask64 m, __m512i a, __m512i b)
VPADDUSW __m512i _mm512_maskz_adds_epu16 (__mmask32 m, __m512i a, __m512i b)
VPADDUSB __m256i _mm256_mask_adds_epu8 (__m256i s, __mmask32 m, __m256i a, __m256i b)
VPADDUSW __m256i _mm256_mask_adds_epu16 (__m256i s, __mmask16 m, __m256i a, __m256i b)
VPADDUSB __m256i _mm256_maskz_adds_epu8 (__mmask32 m, __m256i a, __m256i b)
VPADDUSW __m256i _mm256_maskz_adds_epu16 (__mmask16 m, __m256i a, __m256i b)
VPADDUSB __m128i _mm_mask_adds_epu8 (__m128i s, __mmask16 m, __m128i a, __m128i b)
VPADDUSW __m128i _mm_mask_adds_epu16 (__m128i s, __mmask8 m, __m128i a, __m128i b)
VPADDUSB __m128i _mm_maskz_adds_epu8 (__mmask16 m, __m128i a, __m128i b)
VPADDUSW __m128i _mm_maskz_adds_epu16 (__mmask8 m, __m128i a, __m128i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, "Type E4 Class Exception Conditions."

PALIGNR—Packed Align Right

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 3A 0F /r ib[1] PALIGNR mm1, mm2/m64, imm8	A	V/V	SSSE3	Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in imm8 into mm1.
66 0F 3A 0F /r ib PALIGNR xmm1, xmm2/m128, imm8	A	V/V	SSSE3	Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in imm8 into xmm1.
VEX.128.66.0F3A.WIG 0F /r ib VPALIGNR xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Concatenate xmm2 and xmm3/m128, extract byte aligned result shifted to the right by constant value in imm8 and result is stored in xmm1.
VEX.256.66.0F3A.WIG 0F /r ib VPALIGNR ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX2	Concatenate pairs of 16 bytes in ymm2 and ymm3/m256 into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in imm8 from each intermediate result, and two 16-byte results are stored in ymm1.
EVEX.128.66.0F3A.WIG 0F /r ib VPALIGNR xmm1 {k1}{z}, xmm2, xmm3/m128, imm8	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Concatenate xmm2 and xmm3/m128 into a 32-byte intermediate result, extract byte aligned result shifted to the right by constant value in imm8 and result is stored in xmm1.
EVEX.256.66.0F3A.WIG 0F /r ib VPALIGNR ymm1 {k1}{z}, ymm2, ymm3/m256, imm8	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Concatenate pairs of 16 bytes in ymm2 and ymm3/m256 into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in imm8 from each intermediate result, and two 16-byte results are stored in ymm1.
EVEX.512.66.0F3A.WIG 0F /r ib VPALIGNR zmm1 {k1}{z}, zmm2, zmm3/m512, imm8	C	V/V	AVX512BW OR AVX10.1	Concatenate pairs of 16 bytes in zmm2 and zmm3/m512 into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in imm8 from each intermediate result, and four 16-byte results are stored in zmm1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

(V)PALIGNR concatenates the destination operand (the first operand) and the source operand (the second operand) into an intermediate composite, shifts the composite at byte granularity to the right by a constant immediate, and extracts the right-aligned result into the destination. The first and the second operands can be an MMX, XMM or a YMM register. The immediate value is considered unsigned. Immediate shift counts larger than the 2L (i.e., 32 for 128-bit operands, or 16 for 64-bit operands) produce a zero result. Both operands can be MMX registers, XMM registers or YMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded by VEX/EVEX prefix, use the REX prefix to access additional registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

EVEX.512 encoded version: The first source operand is a ZMM register and contains four 16-byte blocks. The second source operand is a ZMM register or a 512-bit memory location containing four 16-byte block. The destination operand is a ZMM register and contain four 16-byte results. The imm8[7:0] is the common shift count

used for each of the four successive 16-byte block sources. The low 16-byte block of the two source operands produce the low 16-byte result of the destination operand, the high 16-byte block of the two source operands produce the high 16-byte result of the destination operand and so on for the blocks in the middle.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register and contains two 16-byte blocks. The second source operand is a YMM register or a 256-bit memory location containing two 16-byte block. The destination operand is a YMM register and contain two 16-byte results. The imm8[7:0] is the common shift count used for the two lower 16-byte block sources and the two upper 16-byte block sources. The low 16-byte block of the two source operands produce the low 16-byte result of the destination operand, the high 16-byte block of the two source operands produce the high 16-byte result of the destination operand. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

Concatenation is done with 128-bit data in the first and second source operand for both 128-bit and 256-bit instructions. The high 128-bits of the intermediate composite 256-bit result came from the 128-bit data from the first source operand; the low 128-bits of the intermediate result came from the 128-bit data of the second source operand.

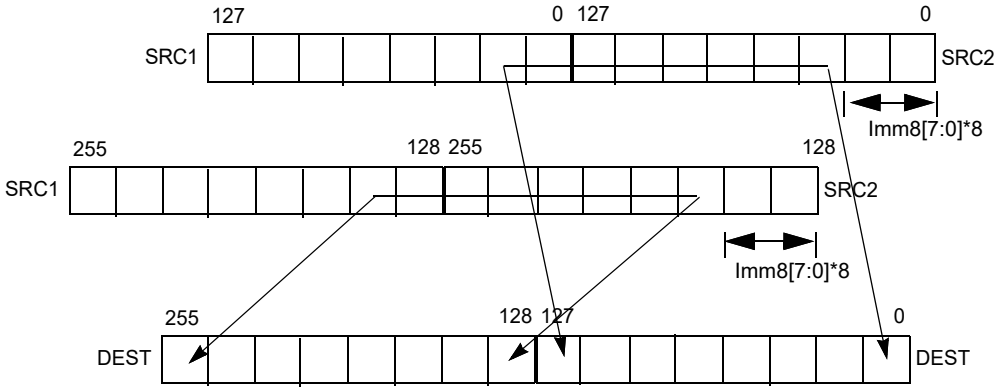


Figure 4-7. 256-bit VPALIGN Instruction Operation

Operation

PALIGNR (With 64-bit Operands)

```
    temp1[127:0] = CONCATENATE(DEST,SRC)>>(imm8*8)
    DEST[63:0] = temp1[63:0]
```

PALIGNR (With 128-bit Operands)

```
temp1[255:0] := ((DEST[127:0] << 128) OR SRC[127:0])>>(imm8*8);
DEST[127:0] := temp1[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

VPALIGNR (VEX.128 Encoded Version)

```
temp1[255:0] := ((SRC1[127:0] << 128) OR SRC2[127:0])>>(imm8*8);
DEST[127:0] := temp1[127:0]
DEST[MAXVL-1:128] := 0
```

VPALIGNR (VEX.256 Encoded Version)

```
temp1[255:0] := ((SRC1[127:0] << 128) OR SRC2[127:0])>>(imm8[7:0]*8);
DEST[127:0] := temp1[127:0]
temp1[255:0] := ((SRC1[255:128] << 128) OR SRC2[255:128])>>(imm8[7:0]*8);
DEST[MAXVL-1:128] := temp1[127:0]
```

VPALIGNR (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR l := 0 TO VL-1 with increments of 128
    temp1[255:0] := ((SRC1[l+127:l] << 128) OR SRC2[l+127:l])>>(imm8[7:0]*8);
    TMP_DEST[l+127:l] := temp1[127:0]
ENDFOR;

FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TMP_DEST[i+7:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PALIGNR __m64 _mm_alignr_pi8 (__m64 a, __m64 b, int n)
(V)PALIGNR __m128i _mm_alignr_epi8 (__m128i a, __m128i b, int n)
VPALIGNR __m256i _mm256_alignr_epi8 (__m256i a, __m256i b, const int n)
VPALIGNR __m512i _mm512_alignr_epi8 (__m512i a, __m512i b, const int n)
VPALIGNR __m512i _mm512_mask_alignr_epi8 (__m512i s, __mmask64 m, __m512i a, __m512i b, const int n)
VPALIGNR __m512i _mm512_maskz_alignr_epi8 (__mmask64 m, __m512i a, __m512i b, const int n)
VPALIGNR __m256i _mm256_mask_alignr_epi8 (__m256i s, __mmask32 m, __m256i a, __m256i b, const int n)
VPALIGNR __m256i _mm256_maskz_alignr_epi8 (__mmask32 m, __m256i a, __m256i b, const int n)
VPALIGNR __m128i _mm_mask_alignr_epi8 (__m128i s, __mmask16 m, __m128i a, __m128i b, const int n)
VPALIGNR __m128i _mm_maskz_alignr_epi8 (__mmask16 m, __m128i a, __m128i b, const int n)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."

EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."

PAND—Logical AND

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DB /r[1] PAND mm, mm/m64	A	V/V	MMX	Bitwise AND mm/m64 and mm.
66 0F DB /r PAND xmm1, xmm2/m128	A	V/V	SSE2	Bitwise AND of xmm2/m128 and xmm1.
VEX.128.66.0F.WIG DB /r VPAND xmm1, xmm2, xmm3/m128	B	V/V	AVX	Bitwise AND of xmm3/m128 and xmm.
VEX.256.66.0F.WIG DB /r VPAND ymm1, ymm2, ymm3/.m256	B	V/V	AVX2	Bitwise AND of ymm2, and ymm3/m256 and store result in ymm1.
EVEX.128.66.0F.W0 DB /r VPANDD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise AND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and store result in xmm1 using writemask k1.
EVEX.256.66.0F.W0 DB /r VPANDD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise AND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and store result in ymm1 using writemask k1.
EVEX.512.66.0F.W0 DB /r VPANDD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Bitwise AND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and store result in zmm1 using writemask k1.
EVEX.128.66.0F.W1 DB /r VPANDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise AND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and store result in xmm1 using writemask k1.
EVEX.256.66.0F.W1 DB /r VPANDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise AND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and store result in ymm1 using writemask k1.
EVEX.512.66.0F.W1 DB /r VPANDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Bitwise AND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and store result in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical AND operation on the first source operand and second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bits of the first and second operands are 1, otherwise it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with write-mask k1 at 32/64-bit granularity.

VEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

Operation

PAND (64-bit Operand)

```
DEST := DEST AND SRC
```

PAND (128-bit Legacy SSE Version)

```
DEST := DEST AND SRC
DEST[MAXVL-1:128] (Unmodified)
```

VPAND (VEX.128 Encoded Version)

```
DEST := SRC1 AND SRC2
DEST[MAXVL-1:128] := 0
```

VPAND (VEX.256 Encoded Instruction)

```
DEST[255:0] := (SRC1[255:0] AND SRC2[255:0])
DEST[MAXVL-1:256] := 0
```

VPANDD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := SRC1[i+31:i] BITWISE AND SRC2[31:0]
                ELSE DEST[i+31:i] := SRC1[i+31:i] BITWISE AND SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPANDQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := SRC1[i+63:i] BITWISE AND SRC2[63:0]
                ELSE DEST[i+63:i] := SRC1[i+63:i] BITWISE AND SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPANDD __m512i _mm512_and_epi32(__m512i a, __m512i b);
VPANDD __m512i _mm512_mask_and_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPANDD __m512i _mm512_maskz_and_epi32(__mmask16 k, __m512i a, __m512i b);
VPANDQ __m512i _mm512_and_epi64(__m512i a, __m512i b);
VPANDQ __m512i _mm512_mask_and_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPANDQ __m512i _mm512_maskz_and_epi64(__mmask8 k, __m512i a, __m512i b);
VPANDND __m256i _mm256_mask_and_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDND __m256i _mm256_maskz_and_epi32(__mmask8 k, __m256i a, __m256i b);
VPANDND __m128i _mm_mask_and_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDND __m128i _mm_maskz_and_epi32(__mmask8 k, __m128i a, __m128i b);
VPANDNQ __m256i _mm256_mask_and_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDNQ __m256i _mm256_maskz_and_epi64(__mmask8 k, __m256i a, __m256i b);
VPANDNQ __m128i _mm_mask_and_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDNQ __m128i _mm_maskz_and_epi64(__mmask8 k, __m128i a, __m128i b);
PAND __m64 _mm_and_si64 (__m64 m1, __m64 m2)
(V)PAND __m128i _mm_and_si128 (__m128i a, __m128i b)
VPAND __m256i _mm256_and_si256 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

PANDN—Logical AND NOT

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DF /r[1] PANDN mm, mm/m64	A	V/V	MMX	Bitwise AND NOT of mm/m64 and mm.
66 0F DF /r PANDN xmm1, xmm2/m128	A	V/V	SSE2	Bitwise AND NOT of xmm2/m128 and xmm1.
VEX.128.66.0F.WIG DF /r VPANDN xmm1, xmm2, xmm3/m128	B	V/V	AVX	Bitwise AND NOT of xmm3/m128 and xmm2.
VEX.256.66.0F.WIG DF /r VPANDN ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Bitwise AND NOT of ymm2, and ymm3/m256 and store result in ymm1.
EVEX.128.66.0F.W0 DF /r VPANDND xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise AND NOT of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and store result in xmm1 using writemask k1.
EVEX.256.66.0F.W0 DF /r VPANDND ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise AND NOT of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and store result in ymm1 using writemask k1.
EVEX.512.66.0F.W0 DF /r VPANDND zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Bitwise AND NOT of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and store result in zmm1 using writemask k1.
EVEX.128.66.0F.W1 DF /r VPANDNQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise AND NOT of packed quadword integers in xmm2 and xmm3/m128/m64bcst and store result in xmm1 using writemask k1.
EVEX.256.66.0F.W1 DF /r VPANDNQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise AND NOT of packed quadword integers in ymm2 and ymm3/m256/m64bcst and store result in ymm1 using writemask k1.
EVEX.512.66.0F.W1 DF /r VPANDNQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Bitwise AND NOT of packed quadword integers in zmm2 and zmm3/m512/m64bcst and store result in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical NOT operation on the first source operand, then performs bitwise AND with second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bit in the first operand is 0 and the corresponding bit in the second operand is 1, otherwise it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with write-mask k1 at 32/64-bit granularity.

VEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

Operation

PANDN (64-bit Operand)

```
DEST := NOT(DEST) AND SRC
```

PANDN (128-bit Legacy SSE Version)

```
DEST := NOT(DEST) AND SRC
DEST[MAXVL-1:128] (Unmodified)
```

VPANDN (VEX.128 Encoded Version)

```
DEST := NOT(SRC1) AND SRC2
DEST[MAXVL-1:128] := 0
```

VPANDN (VEX.256 Encoded Instruction)

```
DEST[255:0] := ((NOT SRC1[255:0]) AND SRC2[255:0])
DEST[MAXVL-1:256] := 0
```

VPANDND (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := ((NOT SRC1[i+31:i]) AND SRC2[31:0])
                ELSE DEST[i+31:i] := ((NOT SRC1[i+31:i]) AND SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPANDNQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := ((NOT SRC1[i+63:i]) AND SRC2[63:0])
                ELSE DEST[i+63:i] := ((NOT SRC1[i+63:i]) AND SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPANDND __m512i _mm512_andnot_epi32(__m512i a, __m512i b);
VPANDND __m512i _mm512_mask_andnot_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPANDND __m512i _mm512_maskz_andnot_epi32(__mmask16 k, __m512i a, __m512i b);
VPANDND __m256i _mm256_mask_andnot_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDND __m256i _mm256_maskz_andnot_epi32(__mmask8 k, __m256i a, __m256i b);
VPANDND __m128i _mm_mask_andnot_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDND __m128i _mm_maskz_andnot_epi32(__mmask8 k, __m128i a, __m128i b);
VPANDNQ __m512i _mm512_andnot_epi64(__m512i a, __m512i b);
VPANDNQ __m512i _mm512_mask_andnot_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPANDNQ __m512i _mm512_maskz_andnot_epi64(__mmask8 k, __m512i a, __m512i b);
VPANDNQ __m256i _mm256_mask_andnot_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDNQ __m256i _mm256_maskz_andnot_epi64(__mmask8 k, __m256i a, __m256i b);
VPANDNQ __m128i _mm_mask_andnot_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDNQ __m128i _mm_maskz_andnot_epi64(__mmask8 k, __m128i a, __m128i b);
PANDN __m64 _mm_andnot_si64 (__m64 m1, __m64 m2)
(V)PANDN __m128i _mm_andnot_si128 (__m128i a, __m128i b)
VPANDN __m256i _mm256_andnot_si256 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

PAUSE—Spin Loop Hint

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 90	PAUSE	ZO	Valid	Valid	Gives hint to processor that improves performance of spin-wait loops.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Improves the performance of spin-wait loops. When executing a "spin-wait loop," processors will suffer a severe performance penalty when exiting the loop because it detects a possible memory order violation. The PAUSE instruction provides a hint to the processor that the code sequence is a spin-wait loop. The processor uses this hint to avoid the memory order violation in most situations, which greatly improves processor performance. For this reason, it is recommended that a PAUSE instruction be placed in all spin-wait loops.

An additional function of the PAUSE instruction is to reduce the power consumed by a processor while executing a spin loop. A processor can execute a spin-wait loop extremely quickly, causing the processor to consume a lot of power while it waits for the resource it is spinning on to become available. Inserting a pause instruction in a spin-wait loop greatly reduces the processor's power consumption.

This instruction was introduced in the Pentium 4 processors, but is backward compatible with all IA-32 processors. In earlier IA-32 processors, the PAUSE instruction operates like a NOP instruction. The Pentium 4 and Intel Xeon processors implement the PAUSE instruction as a delay. The delay is finite and can be zero for some processors. This instruction does not change the architectural state of the processor (that is, it performs essentially a delaying no-op operation).

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
Execute_Next_Instruction(DELAY);
```

Numeric Exceptions

None.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

PAVGB/PAVGW—Average Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E0 /r[1] PAVGB mm1, mm2/m64	A	V/V	SSE	Average packed unsigned byte integers from mm2/m64 and mm1 with rounding.
66 0F E0, /r PAVGB xmm1, xmm2/m128	A	V/V	SSE2	Average packed unsigned byte integers from xmm2/m128 and xmm1 with rounding.
NP 0F E3 /r[1] PAVGW mm1, mm2/m64	A	V/V	SSE	Average packed unsigned word integers from mm2/m64 and mm1 with rounding.
66 0F E3 /r PAVGW xmm1, xmm2/m128	A	V/V	SSE2	Average packed unsigned word integers from xmm2/m128 and xmm1 with rounding.
VEX.128.66.0F.WIG E0 /r VPAVGB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Average packed unsigned byte integers from xmm3/m128 and xmm2 with rounding.
VEX.128.66.0F.WIG E3 /r VPAVGW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Average packed unsigned word integers from xmm3/m128 and xmm2 with rounding.
VEX.256.66.0F.WIG E0 /r VPAVGB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Average packed unsigned byte integers from ymm2, and ymm3/m256 with rounding and store to ymm1.
VEX.256.66.0F.WIG E3 /r VPAVGW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Average packed unsigned word integers from ymm2, ymm3/m256 with rounding to ymm1.
EVEX.128.66.0F.WIG E0 /r VPAVGB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Average packed unsigned byte integers from xmm2, and xmm3/m128 with rounding and store to xmm1 under writemask k1.
EVEX.256.66.0F.WIG E0 /r VPAVGB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Average packed unsigned byte integers from ymm2, and ymm3/m256 with rounding and store to ymm1 under writemask k1.
EVEX.512.66.0F.WIG E0 /r VPAVGB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Average packed unsigned byte integers from zmm2, and zmm3/m512 with rounding and store to zmm1 under writemask k1.
EVEX.128.66.0F.WIG E3 /r VPAVGW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Average packed unsigned word integers from xmm2, xmm3/m128 with rounding to xmm1 under writemask k1.
EVEX.256.66.0F.WIG E3 /r VPAVGW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Average packed unsigned word integers from ymm2, ymm3/m256 with rounding to ymm1 under writemask k1.
EVEX.512.66.0F.WIG E3 /r VPAVGW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Average packed unsigned word integers from zmm2, zmm3/m512 with rounding to zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD average of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the results in the destination operand. For each corresponding pair of data elements in the first and second operands, the elements are added together, a 1 is added to the temporary sum, and that result is shifted right one bit position.

The (V)PAVGB instruction operates on packed unsigned bytes and the (V)PAVGW instruction operates on packed unsigned words.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register or a 512-bit memory location. The destination operand is a ZMM register.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

Operation

PAVGB (With 64-bit Operands)

```
DEST[7:0] := (SRC[7:0] + DEST[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
(* Repeat operation performed for bytes 2 through 6 *)
DEST[63:56] := (SRC[63:56] + DEST[63:56] + 1) >> 1;
```

PAVGW (With 64-bit Operands)

```
DEST[15:0] := (SRC[15:0] + DEST[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
(* Repeat operation performed for words 2 and 3 *)
DEST[63:48] := (SRC[63:48] + DEST[63:48] + 1) >> 1;
```

PAVGB (With 128-bit Operands)

```
DEST[7:0] := (SRC[7:0] + DEST[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
(* Repeat operation performed for bytes 2 through 14 *)
DEST[127:120] := (SRC[127:120] + DEST[127:120] + 1) >> 1;
```

PAVGW (With 128-bit Operands)

```
DEST[15:0] := (SRC[15:0] + DEST[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
(* Repeat operation performed for words 2 through 6 *)
DEST[127:112] := (SRC[127:112] + DEST[127:112] + 1) >> 1;
```

VPAVGB (VEX.128 Encoded Version)

```
    DEST[7:0] := (SRC1[7:0] + SRC2[7:0] + 1) >> 1;
    (* Repeat operation performed for bytes 2 through 15 *)
    DEST[127:120] := (SRC1[127:120] + SRC2[127:120] + 1) >> 1
    DEST[MAXVL-1:128] := 0
```

VPAVGW (VEX.128 Encoded Version)

```
    DEST[15:0] := (SRC1[15:0] + SRC2[15:0] + 1) >> 1;
    (* Repeat operation performed for 16-bit words 2 through 7 *)
    DEST[127:112] := (SRC1[127:112] + SRC2[127:112] + 1) >> 1
    DEST[MAXVL-1:128] := 0
```

VPAVGB (VEX.256 Encoded Instruction)

```
    DEST[7:0] := (SRC1[7:0] + SRC2[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
    (* Repeat operation performed for bytes 2 through 31)
    DEST[255:248] := (SRC1[255:248] + SRC2[255:248] + 1) >> 1;
```

VPAVGW (VEX.256 Encoded Instruction)

```
    DEST[15:0] := (SRC1[15:0] + SRC2[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
    (* Repeat operation performed for words 2 through 15)
    DEST[255:14]) := (SRC1[255:240] + SRC2[255:240] + 1) >> 1;
VPAVGB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := (SRC1[i+7:i] + SRC2[i+7:i] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPAVGW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := (SRC1[i+15:i] + SRC2[i+15:i] + 1) >> 1
                                    ; (* Temp sum before shifting is 17 bits *)
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPAVGB __m512i _mm512_avg_epu8(__m512i a, __m512i b);
VPAVGW __m512i _mm512_avg_epu16(__m512i a, __m512i b);
VPAVGB __m512i _mm512_mask_avg_epu8(__m512i s, __mmask64 m, __m512i a, __m512i b);
VPAVGW __m512i _mm512_mask_avg_epu16(__m512i s, __mmask32 m, __m512i a, __m512i b);
VPAVGB __m512i _mm512_maskz_avg_epu8(__mmask64 m, __m512i a, __m512i b);
VPAVGW __m512i _mm512_maskz_avg_epu16(__mmask32 m, __m512i a, __m512i b);
VPAVGB __m256i _mm256_mask_avg_epu8(__m256i s, __mmask32 m, __m256i a, __m256i b);
VPAVGW __m256i _mm256_mask_avg_epu16(__m256i s, __mmask16 m, __m256i a, __m256i b);
VPAVGB __m256i _mm256_maskz_avg_epu8(__mmask32 m, __m256i a, __m256i b);
VPAVGW __m256i _mm256_maskz_avg_epu16(__mmask16 m, __m256i a, __m256i b);
VPAVGB __m128i _mm_mask_avg_epu8(__m128i s, __mmask16 m, __m128i a, __m128i b);
VPAVGW __m128i _mm_mask_avg_epu16(__m128i s, __mmask8 m, __m128i a, __m128i b);
VPAVGB __m128i _mm_maskz_avg_epu8(__mmask16 m, __m128i a, __m128i b);
VPAVGW __m128i _mm_maskz_avg_epu16(__mmask8 m, __m128i a, __m128i b);
PAVGB __m64 _mm_avg_pu8 (__m64 a, __m64 b)
PAVGW __m64 _mm_avg_pu16 (__m64 a, __m64 b)
(V)PAVGB __m128i _mm_avg_epu8 (__m128i a, __m128i b)
(V)PAVGW __m128i _mm_avg_epu16 (__m128i a, __m128i b)
VPAVGB __m256i _mm256_avg_epu8 (__m256i a, __m256i b)
VPAVGW __m256i _mm256_avg_epu16 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

PBLENDVB—Variable Blend Packed Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 10 /r PBLENDVB xmm1, xmm2/m128, <XMM0>	RM	V/V	SSE4_1	Select byte values from xmm1 and xmm2/m128 from mask specified in the high bit of each byte in XMM0 and store the values into xmm1.
VEX.128.66.0F3A.W0 4C /r /is4 VPBLENDVB xmm1, xmm2, xmm3/m128, xmm4	RVMR	V/V	AVX	Select byte values from xmm2 and xmm3/m128 using mask bits in the specified mask register, xmm4, and store the values into xmm1.
VEX.256.66.0F3A.W0 4C /r /is4 VPBLENDVB ymm1, ymm2, ymm3/m256, ymm4	RVMR	V/V	AVX2	Select byte values from ymm2 and ymm3/m256 from mask specified in the high bit of each byte in ymm4 and store the values into ymm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	<XMM0>	N/A
RVMR	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8[7:4]

Description

Conditionally copies byte elements from the source operand (second operand) to the destination operand (first operand) depending on mask bits defined in the implicit third register argument, XMM0. The mask bits are the most significant bit in each byte element of the XMM0 register.

If a mask bit is "1", then the corresponding byte element in the source operand is copied to the destination, else the byte element in the destination operand is left unchanged.

The register assignment of the implicit third operand is defined to be the architectural register XMM0.

128-bit Legacy SSE version: The first source operand and the destination operand is the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute PBLENDVB with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand is an XMM register or 128-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (MAXVL-1:128) of the corresponding YMM register (destination register) are zeroed. VEX.L must be 0, otherwise the instruction will #UD. VEX.W must be 0, otherwise, the instruction will #UD.

VEX.256 encoded version: The first source operand and the destination operand are YMM registers. The second source operand is an YMM register or 256-bit memory location. The third source register is an YMM register and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored.

VPBLENDVB permits the mask to be any XMM or YMM register. In contrast, PBLENDVB treats XMM0 implicitly as the mask and do not support non-destructive destination operation. An attempt to execute PBLENDVB encoded with a VEX prefix will cause a #UD exception.

Operation

PBLENDVB (128-bit Legacy SSE Version)

```
MASK := XMM0
IF (MASK[7] = 1) THEN DEST[7:0] := SRC[7:0];
ELSE DEST[7:0] := DEST[7:0];
IF (MASK[15] = 1) THEN DEST[15:8] := SRC[15:8];
ELSE DEST[15:8] := DEST[15:8];
IF (MASK[23] = 1) THEN DEST[23:16] := SRC[23:16]
```

```
ELSE DEST[23:16] := DEST[23:16];
IF (MASK[31] = 1) THEN DEST[31:24] := SRC[31:24]
ELSE DEST[31:24] := DEST[31:24];
IF (MASK[39] = 1) THEN DEST[39:32] := SRC[39:32]
ELSE DEST[39:32] := DEST[39:32];
IF (MASK[47] = 1) THEN DEST[47:40] := SRC[47:40]
ELSE DEST[47:40] := DEST[47:40];
IF (MASK[55] = 1) THEN DEST[55:48] := SRC[55:48]
ELSE DEST[55:48] := DEST[55:48];
IF (MASK[63] = 1) THEN DEST[63:56] := SRC[63:56]
ELSE DEST[63:56] := DEST[63:56];
IF (MASK[71] = 1) THEN DEST[71:64] := SRC[71:64]
ELSE DEST[71:64] := DEST[71:64];
IF (MASK[79] = 1) THEN DEST[79:72] := SRC[79:72]
ELSE DEST[79:72] := DEST[79:72];
IF (MASK[87] = 1) THEN DEST[87:80] := SRC[87:80]
ELSE DEST[87:80] := DEST[87:80];
IF (MASK[95] = 1) THEN DEST[95:88] := SRC[95:88]
ELSE DEST[95:88] := DEST[95:88];
IF (MASK[103] = 1) THEN DEST[103:96] := SRC[103:96]
ELSE DEST[103:96] := DEST[103:96];
IF (MASK[111] = 1) THEN DEST[111:104] := SRC[111:104]
ELSE DEST[111:104] := DEST[111:104];
IF (MASK[119] = 1) THEN DEST[119:112] := SRC[119:112]
ELSE DEST[119:112] := DEST[119:112];
IF (MASK[127] = 1) THEN DEST[127:120] := SRC[127:120]
ELSE DEST[127:120] := DEST[127:120])
DEST[MAXVL-1:128] (Unmodified)
```

VPBLENDVB (VEX.128 Encoded Version)

```
MASK := SRC3
IF (MASK[7] = 1) THEN DEST[7:0] := SRC2[7:0];
ELSE DEST[7:0] := SRC1[7:0];
IF (MASK[15] = 1) THEN DEST[15:8] := SRC2[15:8];
ELSE DEST[15:8] := SRC1[15:8];
IF (MASK[23] = 1) THEN DEST[23:16] := SRC2[23:16]
ELSE DEST[23:16] := SRC1[23:16];
IF (MASK[31] = 1) THEN DEST[31:24] := SRC2[31:24]
ELSE DEST[31:24] := SRC1[31:24];
IF (MASK[39] = 1) THEN DEST[39:32] := SRC2[39:32]
ELSE DEST[39:32] := SRC1[39:32];
IF (MASK[47] = 1) THEN DEST[47:40] := SRC2[47:40]
ELSE DEST[47:40] := SRC1[47:40];
IF (MASK[55] = 1) THEN DEST[55:48] := SRC2[55:48]
ELSE DEST[55:48] := SRC1[55:48];
IF (MASK[63] = 1) THEN DEST[63:56] := SRC2[63:56]
ELSE DEST[63:56] := SRC1[63:56];
IF (MASK[71] = 1) THEN DEST[71:64] := SRC2[71:64]
ELSE DEST[71:64] := SRC1[71:64];
IF (MASK[79] = 1) THEN DEST[79:72] := SRC2[79:72]
ELSE DEST[79:72] := SRC1[79:72];
IF (MASK[87] = 1) THEN DEST[87:80] := SRC2[87:80]
ELSE DEST[87:80] := SRC1[87:80];
IF (MASK[95] = 1) THEN DEST[95:88] := SRC2[95:88]
```

ELSE DEST[95:88] := SRC1[95:88];
IF (MASK[103] = 1) THEN DEST[103:96] := SRC2[103:96]
ELSE DEST[103:96] := SRC1[103:96];
IF (MASK[111] = 1) THEN DEST[111:104] := SRC2[111:104]
ELSE DEST[111:104] := SRC1[111:104];
IF (MASK[119] = 1) THEN DEST[119:112] := SRC2[119:112]
ELSE DEST[119:112] := SRC1[119:112];
IF (MASK[127] = 1) THEN DEST[127:120] := SRC2[127:120]
ELSE DEST[127:120] := SRC1[127:120])
DEST[MAXVL-1:128] := 0

VPBLENDVB (VEX.256 Encoded Version)

MASK := SRC3
IF (MASK[7] == 1) THEN DEST[7:0] := SRC2[7:0];
ELSE DEST[7:0] := SRC1[7:0];
IF (MASK[15] == 1) THEN DEST[15:8] := SRC2[15:8];
ELSE DEST[15:8] := SRC1[15:8];
IF (MASK[23] == 1) THEN DEST[23:16] := SRC2[23:16]
ELSE DEST[23:16] := SRC1[23:16];
IF (MASK[31] == 1) THEN DEST[31:24] := SRC2[31:24]
ELSE DEST[31:24] := SRC1[31:24];
IF (MASK[39] == 1) THEN DEST[39:32] := SRC2[39:32]
ELSE DEST[39:32] := SRC1[39:32];
IF (MASK[47] == 1) THEN DEST[47:40] := SRC2[47:40]
ELSE DEST[47:40] := SRC1[47:40];
IF (MASK[55] == 1) THEN DEST[55:48] := SRC2[55:48]
ELSE DEST[55:48] := SRC1[55:48];
IF (MASK[63] == 1) THEN DEST[63:56] := SRC2[63:56]
ELSE DEST[63:56] := SRC1[63:56];
IF (MASK[71] == 1) THEN DEST[71:64] := SRC2[71:64]
ELSE DEST[71:64] := SRC1[71:64];
IF (MASK[79] == 1) THEN DEST[79:72] := SRC2[79:72]
ELSE DEST[79:72] := SRC1[79:72];
IF (MASK[87] == 1) THEN DEST[87:80] := SRC2[87:80]
ELSE DEST[87:80] := SRC1[87:80];
IF (MASK[95] == 1) THEN DEST[95:88] := SRC2[95:88]
ELSE DEST[95:88] := SRC1[95:88];
IF (MASK[103] == 1) THEN DEST[103:96] := SRC2[103:96]
ELSE DEST[103:96] := SRC1[103:96];
IF (MASK[111] == 1) THEN DEST[111:104] := SRC2[111:104]
ELSE DEST[111:104] := SRC1[111:104];
IF (MASK[119] == 1) THEN DEST[119:112] := SRC2[119:112]
ELSE DEST[119:112] := SRC1[119:112];
IF (MASK[127] == 1) THEN DEST[127:120] := SRC2[127:120]
ELSE DEST[127:120] := SRC1[127:120])
IF (MASK[135] == 1) THEN DEST[135:128] := SRC2[135:128];
ELSE DEST[135:128] := SRC1[135:128];
IF (MASK[143] == 1) THEN DEST[143:136] := SRC2[143:136];
ELSE DEST[[143:136] := SRC1[143:136];
IF (MASK[151] == 1) THEN DEST[151:144] := SRC2[151:144]
ELSE DEST[151:144] := SRC1[151:144];
IF (MASK[159] == 1) THEN DEST[159:152] := SRC2[159:152]
ELSE DEST[159:152] := SRC1[159:152];
IF (MASK[167] == 1) THEN DEST[167:160] := SRC2[167:160]

```
ELSE DEST[167:160] := SRC1[167:160];
IF (MASK[175] == 1) THEN DEST[175:168] := SRC2[175:168]
ELSE DEST[175:168] := SRC1[175:168];
IF (MASK[183] == 1) THEN DEST[183:176] := SRC2[183:176]
ELSE DEST[183:176] := SRC1[183:176];
IF (MASK[191] == 1) THEN DEST[191:184] := SRC2[191:184]
ELSE DEST[191:184] := SRC1[191:184];
IF (MASK[199] == 1) THEN DEST[199:192] := SRC2[199:192]
ELSE DEST[199:192] := SRC1[199:192];
IF (MASK[207] == 1) THEN DEST[207:200] := SRC2[207:200]
ELSE DEST[207:200] := SRC1[207:200]
IF (MASK[215] == 1) THEN DEST[215:208] := SRC2[215:208]
ELSE DEST[215:208] := SRC1[215:208];
IF (MASK[223] == 1) THEN DEST[223:216] := SRC2[223:216]
ELSE DEST[223:216] := SRC1[223:216];
IF (MASK[231] == 1) THEN DEST[231:224] := SRC2[231:224]
ELSE DEST[231:224] := SRC1[231:224];
IF (MASK[239] == 1) THEN DEST[239:232] := SRC2[239:232]
ELSE DEST[239:232] := SRC1[239:232];
IF (MASK[247] == 1) THEN DEST[247:240] := SRC2[247:240]
ELSE DEST[247:240] := SRC1[247:240];
IF (MASK[255] == 1) THEN DEST[255:248] := SRC2[255:248]
ELSE DEST[255:248] := SRC1[255:248]
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PBLENDVB __m128i _mm_blendv_epi8 (__m128i v1, __m128i v2, __m128i mask);
VPBLENDVB __m256i _mm256_blendv_epi8 (__m256i v1, __m256i v2, __m256i mask);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions," additionally:

#UD	If VEX.W = 1.

PBLENDW—Blend Packed Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 0E /r ib PBLENDW xmm1, xmm2/m128, imm8	RMI	V/V	SSE4_1	Select words from xmm1 and xmm2/m128 from mask specified in imm8 and store the values into xmm1.
VEX.128.66.0F3A.WIG 0E /r ib VPBLENDW xmm1, xmm2, xmm3/m128, imm8	RVMI	V/V	AVX	Select words from xmm2 and xmm3/m128 from mask specified in imm8 and store the values into xmm1.
VEX.256.66.0F3A.WIG 0E /r ib VPBLENDW ymm1, ymm2, ymm3/m256, imm8	RVMI	V/V	AVX2	Select words from ymm2 and ymm3/m256 from mask specified in imm8 and store the values into ymm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Words from the source operand (second operand) are conditionally written to the destination operand (first operand) depending on bits in the immediate operand (third operand). The immediate bits (bits 7:0) form a mask that determines whether the corresponding word in the destination is copied from the source. If a bit in the mask, corresponding to a word, is "1", then the word is copied, else the word element in the destination operand is unchanged.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

PBLENDW (128-bit Legacy SSE Version)

```
IF (imm8[0] = 1) THEN DEST[15:0] := SRC[15:0]
ELSE DEST[15:0] := DEST[15:0]
IF (imm8[1] = 1) THEN DEST[31:16] := SRC[31:16]
ELSE DEST[31:16] := DEST[31:16]
IF (imm8[2] = 1) THEN DEST[47:32] := SRC[47:32]
ELSE DEST[47:32] := DEST[47:32]
IF (imm8[3] = 1) THEN DEST[63:48] := SRC[63:48]
ELSE DEST[63:48] := DEST[63:48]
IF (imm8[4] = 1) THEN DEST[79:64] := SRC[79:64]
ELSE DEST[79:64] := DEST[79:64]
IF (imm8[5] = 1) THEN DEST[95:80] := SRC[95:80]
ELSE DEST[95:80] := DEST[95:80]
IF (imm8[6] = 1) THEN DEST[111:96] := SRC[111:96]
ELSE DEST[111:96] := DEST[111:96]
IF (imm8[7] = 1) THEN DEST[127:112] := SRC[127:112]
```

ELSE DEST[127:112] := DEST[127:112]

VPBLENDW (VEX.128 Encoded Version)

```
IF (imm8[0] = 1) THEN DEST[15:0] := SRC2[15:0]
ELSE DEST[15:0] := SRC1[15:0]
IF (imm8[1] = 1) THEN DEST[31:16] := SRC2[31:16]
ELSE DEST[31:16] := SRC1[31:16]
IF (imm8[2] = 1) THEN DEST[47:32] := SRC2[47:32]
ELSE DEST[47:32] := SRC1[47:32]
IF (imm8[3] = 1) THEN DEST[63:48] := SRC2[63:48]
ELSE DEST[63:48] := SRC1[63:48]
IF (imm8[4] = 1) THEN DEST[79:64] := SRC2[79:64]
ELSE DEST[79:64] := SRC1[79:64]
IF (imm8[5] = 1) THEN DEST[95:80] := SRC2[95:80]
ELSE DEST[95:80] := SRC1[95:80]
IF (imm8[6] = 1) THEN DEST[111:96] := SRC2[111:96]
ELSE DEST[111:96] := SRC1[111:96]
IF (imm8[7] = 1) THEN DEST[127:112] := SRC2[127:112]
ELSE DEST[127:112] := SRC1[127:112]
DEST[MAXVL-1:128] := 0
```

VPBLENDW (VEX.256 Encoded Version)

```
IF (imm8[0] == 1) THEN DEST[15:0] := SRC2[15:0]
ELSE DEST[15:0] := SRC1[15:0]
IF (imm8[1] == 1) THEN DEST[31:16] := SRC2[31:16]
ELSE DEST[31:16] := SRC1[31:16]
IF (imm8[2] == 1) THEN DEST[47:32] := SRC2[47:32]
ELSE DEST[47:32] := SRC1[47:32]
IF (imm8[3] == 1) THEN DEST[63:48] := SRC2[63:48]
ELSE DEST[63:48] := SRC1[63:48]
IF (imm8[4] == 1) THEN DEST[79:64] := SRC2[79:64]
ELSE DEST[79:64] := SRC1[79:64]
IF (imm8[5] == 1) THEN DEST[95:80] := SRC2[95:80]
ELSE DEST[95:80] := SRC1[95:80]
IF (imm8[6] == 1) THEN DEST[111:96] := SRC2[111:96]
ELSE DEST[111:96] := SRC1[111:96]
IF (imm8[7] == 1) THEN DEST[127:112] := SRC2[127:112]
ELSE DEST[127:112] := SRC1[127:112]
IF (imm8[0] == 1) THEN DEST[143:128] := SRC2[143:128]
ELSE DEST[143:128] := SRC1[143:128]
IF (imm8[1] == 1) THEN DEST[159:144] := SRC2[159:144]
ELSE DEST[159:144] := SRC1[159:144]
IF (imm8[2] == 1) THEN DEST[175:160] := SRC2[175:160]
ELSE DEST[175:160] := SRC1[175:160]
IF (imm8[3] == 1) THEN DEST[191:176] := SRC2[191:176]
ELSE DEST[191:176] := SRC1[191:176]
IF (imm8[4] == 1) THEN DEST[207:192] := SRC2[207:192]
ELSE DEST[207:192] := SRC1[207:192]
IF (imm8[5] == 1) THEN DEST[223:208] := SRC2[223:208]
ELSE DEST[223:208] := SRC1[223:208]
IF (imm8[6] == 1) THEN DEST[239:224] := SRC2[239:224]
ELSE DEST[239:224] := SRC1[239:224]
IF (imm8[7] == 1) THEN DEST[255:240] := SRC2[255:240]
ELSE DEST[255:240] := SRC1[255:240]
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PBLENDW __m128i _mm_blend_epi16 (__m128i v1, __m128i v2, const int mask);
VPBLENDW __m256i _mm256_blend_epi16 (__m256i v1, __m256i v2, const int mask)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions," additionally:

#UD	If VEX.L = 1 and AVX2 = 0.

PBNDKB—Platform Bind Key to Binary Large Object

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 C7 PBNDKB	ZO	V/I	PBNDKB	This instruction is used to bind information to a platform by encrypting it with a platform-specific wrapping key.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

The PBNDKB instruction allows software to bind information to a platform by encrypting it with a platform-specific wrapping key. The encrypted data may later be used by the PCONFIG instruction to configure the total storage encryption (TSE) engine.

The instruction can be executed only in 64-bit mode. The registers RBX and RCX provide input information to the instruction. Executions of PBNDKB may fail for platform-specific reasons. An execution reports failure by setting the ZF flag and loading EAX with a non-zero failure reason; a successful execution clears ZF and EAX.

The instruction operates on 256-byte data structures called **bind structures**. It reads a bind structure at the linear address in RBX and writes a modified bind structure to the linear address in RCX. The addresses in RBX and RCX must be different from each other and must be 256-byte aligned.

The instruction encrypts a portion of the input bind structure and generates a MAC of parts of that structure. The encrypted data and MAC are written out as part of the output bind structure.

The format of a bind structure is given in Table 4-13.

Table 4-13. Bind Structure Format

Field	Offset (bytes)	Size (bytes)	Comments
MAC	0	16	Output by PBNDKB as a MAC based on the input bind structure
Reserved	16	8	Reserved; must be zero on input, output as zero
IV	24	12	Initialization vector generated and output by PBNDKB
Reserved	36	28	Reserved; must be zero on input, output as zero
BTENCDATA	64	64	Encryption data (plaintext on input; ciphertext on output)
BTDATA	128	128	Additional control and data (modified but not encrypted)

A description of each of the fields in a bind structure is provided below:

- **MAC:** A MAC produced by PBNDKB of parts of its input bind structure. This field in the input bind structure is not used.
- **IV:** PBNDKB randomly generates a 96-bit initialization vector and uses it as input to an authenticated encryption function. The generated IV is written to the output bind structure. If there is insufficient entropy for the random-number generator, PBNDKB will fail and report the failure by loading EAX with value 1 (ENTROPY_ERROR). This field in the input bind structure is not used.
- **BTENCDATA:** In the input bind structure, the field contains the data to be encrypted. The data consist of two 256-bit keys, a data key and a tweak key. If the value of the KEY_GENERATION_CTRL field of the BTDATA (see below) is 1, PBNDKB randomizes the values of these keys before encrypting them. (If there is insufficient entropy for the random-number generator, PBNDKB will fail and report the failure by loading EAX with value 1 (ENTROPY_ERROR).) PBNDKB writes the encrypted data to this field in the output bind structure.
- **BTDATA:** This field contains additional control and data that are not encrypted. It has the following format:

— **USER_SUPP_CHALLENGE** (bytes 31:0): PBNDKB uses this value in the input bind structure to determine the wrapping key (see below). It writes zero to this field in the output bind structure.

— **KEY_GENERATION_CTRL** (byte 32): PBNDKB uses this value in the input bind structure to determine whether to randomize the keys being encrypted. The value must be 0 or 1 (otherwise, a #GP occurs).

— The remaining 95 bytes are reserved and must be zero.

PBNDKB determines a 256-bit **wrapping key** by computing an HMAC based on SHA-256 using 256-bit platform-specific key and the USER_SUPP_CHALLENGE in the BTDATA field in the input bind structure.

PBNDKB then uses the wrapping key and an AES GCM authenticated encryption function to encrypt BTENCDATA and produce a MAC. The encryption function uses the following inputs:

- The 64-byte BTENCDATA to be encrypted (which may have been randomized; see above).
- The 256-bit wrapping key.
- The 96-bit IV randomly generated by PBNDKB.
- 176 bytes of additional authenticated data that are the concatenation of 8 bytes of zeroes, the IV, 28 bytes of zeroes, and the BTDATA in the input bind structure.
- The length of the additional authenticated data (176).

The encryption function produces a structure with 64 bytes of encrypted data and a 16-byte MAC. PBNDKB saves these values to the corresponding fields in its output bind structure. Other fields are copied from the input bind structure or written as zero, except the IV (which receives the randomly generated value) and the USER_SUPP_CHALLENGE in the BTDATA, which is written as zero.

Operation

```
(* #UD if PBNDKB is not enumerated, CPL > 0, or not in 64-bit mode*)
IF CPUID.07H.01H:EBX.PBNDKB[bit 1] = 0 OR CPL > 0 OR not in 64-bit mode
    THEN #UD; FI;

(* #GP if pointers are not aligned or overlapping *)
IF RBX = RCX OR RBX is not 256-byte aligned OR RCX is not 256-byte aligned
    THEN #GP(0); FI;

Load TMP_BIND_STRUCT from 256 bytes at linear address in RBX;
(* MAC and IV fields might not be read. *)

(* Check TMP_BIND_STRUCT for illegal values *)
IF bytes 23:16 and bytes 63:36 of TMP_BIND_STRUCT are not all zero
    THEN #GP(0); FI;
IF TMP_BIND_STRUCT.BTDATA.KEY_GENERATION_CTRL > 1
    THEN #GP(0); FI;
IF bytes 127:33 of TMP_BIND_STRUCT.BTDATA are not all zero
    THEN #GP(0); FI;

(* Randomize input keys if requested *)
IF TMP_BIND_STRUCT.BTDATA.KEY_GENERATION_CONTROL= 1
    THEN
        Load RNG_DATA_KEY with a random 256-bit value using hardware RNG;
        Load RNG_TWEAK_KEY with a random 256-bit value using hardware RNG;
        IF there was insufficient entropy
            THEN (* PBNDKB failure *)
                RFLAGS.ZF := 1;
                RAX := ENTROPY_ERROR;    (* failure reason 1 *)
            GOTO EXIT;
        FI;
```

```
        (* XOR the input keys with the random keys; this does not modify input bind structure in memory *)
        TMP_BIND_STRUCT.BTENCDATA.DATA_KEY := RNG_DATA_KEY XOR TMP_BIND_STRUCT.BTENCDATA.DATA_KEY;
        TMP_BIND_STRUCT.BTENCDATA.TWEAK_KEY := RNG_TWEAK_KEY XOR TMP_BIND_STRUCT.BTENCDATA.TWEAK_KEY;
FI;

(* Compute wrapping key from platform key and user challenge *)
PLATFORM_KEY := 256-bit platform-specific key;
WRAPPING_KEY := HMAC_SHA256(PLATFORM_KEY, TMP_BIND_STRUCT.BTDATA.USER_SUPP_CHALLENGE);

(* Generate random data for initialization vector *)
Load TMP_IV with a random 96-bit value using hardware RNG;
IF there was insufficient entropy
    THEN (* PBNDKB failure *)
        RFLAGS.ZF := 1;
        RAX := ENTROPY_ERROR;    (* failure reason 1 *)
        GOTO EXIT;
FI;

(* Compose 176 bytes of additional authenticated data for use by authenticated decryption *)
AAD := Concatenation of 8 bytes of zeroes, TMP_IV, 28 bytes of zeroes, and TMP_BIND_STRUCT.BTDATA;

ENCRYPT_STRUCT := AES256_GCM_ENC(TMP_BIND_STRUCT.BTENCDATA, WRAPPING_KEY, TMP_IV, AAD, 176);

OUT_BIND_STRUCT.MAC := ENCRYPT_STRUCT.MAC;
OUT_BIND_STRUCT[bytes 23:16] := 0;
OUT_BIND_STRUCT.IV := TMP_IV;
OUT_BIND_STRUCT[bytes 63:36] := 0;
OUT_BIND_STRUCT.BTENCDATA := ENCRYPT_STRUCT.ENC_DATA;
OUT_BIND_STRUCT.BTDATA.USER_SUPP_CHALLENGE := 0;
OUT_BIND_STRUCT.BTDATA.KEY_GENERATION_CTRL := IN_BIND_STRUCT.BTDATA.KEY_GENERATION_CTRL;
OUT_BIND_STRUCT.BTDATA[bytes 127:33]  := 0;

(* Save OUT_BIND_STRUCT to memory *)
Store OUT_BIND_STRUCT to 256 bytes at linear address in RCX;

(* Indicate successful completion *)
RAX := 0;
RFLAGS.ZF := 0;

EXIT:
RFLAGS.CF := 0;
RFLAGS.PF := 0;
RFLAGS.AF := 0;
RFLAGS.OF := 0;
RFLAGS.SF := 0;
```

Protected Mode Exceptions

#UD	PBNDKB is not supported in protected mode.

Real-Address Mode Exceptions

#UD	PBNDKB is not supported in real-address mode.

Virtual-8086 Mode Exceptions

#UD	PBNDKB is not supported in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	PBNDKB is not supported in compatibility mode.

64-Bit Mode Exceptions

#GP(0)	If the values of RBX and RCX are not both canonical.
	If RBX or RCX is not 256B aligned.
	If RBX = RCX.
	If any of the reserved bytes in the input bind structure are set (including bytes in BTDATA).
	If the value of the key-generation control in the BTDATA field of the input bind structure is not 0 or 1.
#PF(fault-code)	If a page fault occurs in accessing memory operands.
#UD	If the LOCK prefix are used.
	If the current privilege level is not 0.
	If CPUID.07H.01H:EBX.PBNDKB[1] = 0.

PCLMULQDQ—Carry-Less Multiplication Quadword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 44 /r ib PCLMULQDQ xmm1, xmm2/m128, imm8	A	V/V	PCLMULQDQ	Performs carry-less multiplication of one quadword of xmm1 by one quadword of xmm2/m128, stores the 128-bit result in xmm1. The immediate is used to determine which quadwords of xmm1 and xmm2/m128 should be used.
VEX.128.66.0F3A.WIG 44 /r ib VPCLMULQDQ xmm1, xmm2, xmm3/m128, imm8	B	V/V	PCLMULQDQ AVX	Performs carry-less multiplication of one quadword of xmm2 by one quadword of xmm3/m128, stores the 128-bit result in xmm1. The immediate is used to determine which quadwords of xmm2 and xmm3/m128 should be used.
VEX.256.66.0F3A.WIG 44 /r /ib VPCLMULQDQ ymm1, ymm2, ymm3/m256, imm8	B	V/V	VPCLMULQDQ AVX	For each 128-bit lane, performs two carry-less multiplications of one quadword of ymm2 by one quadword of ymm3/m256, stores the two 128-bit results in ymm1. The immediate is used to determine which quadword in each 128-bit lane of ymm2 and ymm3/m256 should be used.
EVEX.128.66.0F3A.WIG 44 /r /ib VPCLMULQDQ xmm1, xmm2, xmm3/m128, imm8	C	V/V	VPCLMULQDQ (AVX512VL OR AVX10.1)	Performs carry-less multiplication of one quadword of xmm2 by one quadword of xmm3/m128, stores the 128-bit result in xmm1. The immediate is used to determine which quadwords of xmm2 and xmm3/m128 should be used.
EVEX.256.66.0F3A.WIG 44 /r /ib VPCLMULQDQ ymm1, ymm2, ymm3/m256, imm8	C	V/V	VPCLMULQDQ (AVX512VL OR AVX10.1)	For each 128-bit lane, performs two carry-less multiplications of one quadword of ymm2 by one quadword of ymm3/m256, stores the two 128-bit results in ymm1. The immediate is used to determine which quadword in each 128-bit lane of ymm2 and ymm3/m256 should be used.
EVEX.512.66.0F3A.WIG 44 /r /ib VPCLMULQDQ zmm1, zmm2, zmm3/m512, imm8	C	V/V	VPCLMULQDQ (AVX512F OR AVX10.1)	For each 128-bit lane, performs two carry-less multiplications of one quadword of zmm2 by one quadword of zmm3/m512, stores the four 128-bit results in zmm1. The immediate is used to determine which quadword in each 128-bit lane of zmm2 and zmm3/m512 should be used.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

Performs packed carry-less multiplication of quadword pairs. XMM versions perform a single multiply of a pair of

quadwords. YMM versions perform two packed multiplies of pairs of quadwords. ZMM versions perform four packed multiplies of pairs of quadwords. Bits 4 and 0 are used to select which 64-bit half of each operand to use according to Table 4-14, other bits of the immediate byte are ignored.

The EVEX encoded form of this instruction does not support memory fault suppression.

Table 4-14. PCLMULQDQ Quadword Selection of Immediate Byte

Imm[4]	Imm[0]	PCLMULQDQ Operation
0	0	CL_MUL(SRC2[1][63:0], SRC1[63:0])
0	1	CL_MUL(SRC2[63:0], SRC1[127:64])
1	0	CL_MUL(SRC2[127:64], SRC1[63:0])
1	1	CL_MUL(SRC2[127:64], SRC1[127:64])

NOTES:

1. SRC2 denotes the second source operand, which can be a register or memory; SRC1 denotes the first source and destination operand.

The first source operand and the destination operand are the same and must be a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. Bits (VL_MAX-1:128) of the corresponding YMM destination register remain unchanged.

Compilers and assemblers may implement the following pseudo-op syntax to simplify programming and emit the required encoding for imm8.

Table 4-15. Pseudo-Op and PCLMULQDQ Implementation

Pseudo-Op	Imm8 Encoding
PCLMULLQLQDQ xmm1, xmm2	**0000_0000B**
PCLMULHQLQDQ xmm1, xmm2	**0000_0001B**
PCLMULLQHQDQ xmm1, xmm2	**0001_0000B**
PCLMULHQHQDQ xmm1, xmm2	**0001_0001B**

Operation

```
define PCLMUL128(X,Y):                    // helper function
    FOR i := 0 to 63:
        TMP [ i ] := X[ 0 ] and Y[ i ]
        FOR j := 1 to i:
            TMP [ i ] := TMP [ i ] xor (X[ j ] and Y[ i - j ])
        DEST[ i ] := TMP[ i ]
    FOR i := 64 to 126:
        TMP [ i ] := 0
        FOR j := i - 63 to 63:
            TMP [ i ] := TMP [ i ] xor (X[ j ] and Y[ i - j ])
        DEST[ i ] := TMP[ i ]
    DEST[127] := 0;
    RETURN DEST                           // 128b vector
```

PCLMULQDQ (SSE Version)

```
IF imm8[0] = 0:
    TEMP1 := SRC1.qword[0]
ELSE:
    TEMP1 := SRC1.qword[1]
IF imm8[4] = 0:
    TEMP2 := SRC2.qword[0]
ELSE:
    TEMP2 := SRC2.qword[1]
DEST[127:0] := PCLMUL128(TEMP1, TEMP2)
DEST[MAXVL-1:128] (Unmodified)
```

VPCLMULQDQ (128b and 256b VEX Encoded Versions)

```
(KL,VL) = (1,128), (2,256)
FOR i= 0 to KL-1:
    IF imm8[0] = 0:
        TEMP1 := SRC1.xmm[i].qword[0]
    ELSE:
        TEMP1 := SRC1.xmm[i].qword[1]
    IF imm8[4] = 0:
        TEMP2 := SRC2.xmm[i].qword[0]
    ELSE:
        TEMP2 := SRC2.xmm[i].qword[1]
    DEST.xmm[i] := PCLMUL128(TEMP1, TEMP2)
DEST[MAXVL-1:VL] := 0
```

VPCLMULQDQ (EVEX Encoded Version)

```
(KL,VL) = (1,128), (2,256), (4,512)
FOR i = 0 to KL-1:
    IF imm8[0] = 0:
        TEMP1 := SRC1.xmm[i].qword[0]
    ELSE:
        TEMP1 := SRC1.xmm[i].qword[1]
    IF imm8[4] = 0:
        TEMP2 := SRC2.xmm[i].qword[0]
    ELSE:
        TEMP2 := SRC2.xmm[i].qword[1]
    DEST.xmm[i] := PCLMUL128(TEMP1, TEMP2)
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PCLMULQDQ __m128i _mm_clmulepi64_si128 (__m128i, __m128i, const int)
VPCLMULQDQ __m256i _mm256_clmulepi64_epi128(__m256i, __m256i, const int);
VPCLMULQDQ __m512i _mm512_clmulepi64_epi128(__m512i, __m512i, const int);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions," additionally:

#UD	If VEX.L = 1.

EVEX-encoded: See Table 2-52, "Type E4NF Class Exception Conditions."

PCMPEQB/PCMPEQW/PCMPEQD— Compare Packed Data for Equal

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 74 /r[1] PCMPEQB mm, mm/m64	A	V/V	MMX	Compare packed bytes in mm/m64 and mm for equality.
66 0F 74 /r PCMPEQB xmm1, xmm2/m128	A	V/V	SSE2	Compare packed bytes in xmm2/m128 and xmm1 for equality.
NP 0F 75 /r[1] PCMPEQW mm, mm/m64	A	V/V	MMX	Compare packed words in mm/m64 and mm for equality.
66 0F 75 /r PCMPEQW xmm1, xmm2/m128	A	V/V	SSE2	Compare packed words in xmm2/m128 and xmm1 for equality.
NP 0F 76 /r[1] PCMPEQD mm, mm/m64	A	V/V	MMX	Compare packed doublewords in mm/m64 and mm for equality.
66 0F 76 /r PCMPEQD xmm1, xmm2/m128	A	V/V	SSE2	Compare packed doublewords in xmm2/m128 and xmm1 for equality.
VEX.128.66.0F.WIG 74 /r VPCMPEQB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed bytes in xmm3/m128 and xmm2 for equality.
VEX.128.66.0F.WIG 75 /r VPCMPEQW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed words in xmm3/m128 and xmm2 for equality.
VEX.128.66.0F.WIG 76 /r VPCMPEQD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed doublewords in xmm3/m128 and xmm2 for equality.
VEX.256.66.0F.WIG 74 /r VPCMPEQB ymm1, ymm2, ymm3 /m256	B	V/V	AVX2	Compare packed bytes in ymm3/m256 and ymm2 for equality.
VEX.256.66.0F.WIG 75 /r VPCMPEQW ymm1, ymm2, ymm3 /m256	B	V/V	AVX2	Compare packed words in ymm3/m256 and ymm2 for equality.
VEX.256.66.0F.WIG 76 /r VPCMPEQD ymm1, ymm2, ymm3 /m256	B	V/V	AVX2	Compare packed doublewords in ymm3/m256 and ymm2 for equality.
EVEX.128.66.0F.W0 76 /r VPCMPEQD k1 {k2}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare Equal between int32 vector xmm2 and int32 vector xmm3/m128/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F.W0 76 /r VPCMPEQD k1 {k2}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare Equal between int32 vector ymm2 and int32 vector ymm3/m256/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F.W0 76 /r VPCMPEQD k1 {k2}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Compare Equal between int32 vectors in zmm2 and zmm3/m512/m32bcst, and set destination k1 according to the comparison results under writemask k2.
EVEX.128.66.0F.WIG 74 /r VPCMPEQB k1 {k2}, xmm2, xmm3 /m128	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed bytes in xmm3/m128 and xmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F.WIG 74 /r VPCMPEQB k1 {k2}, ymm2, ymm3 /m256	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed bytes in ymm3/m256 and ymm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F.WIG 74 /r VPCMPEQB k1 {k2}, zmm2, zmm3 /m512	D	V/V	AVX512BW OR AVX10.1	Compare packed bytes in zmm3/m512 and zmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.128.66.0F.WIG 75 /r VPCMPEQW k1 {k2}, xmm2, xmm3 /m128	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed words in xmm3/m128 and xmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F.WIG 75 /r VPCMPEQW k1 {k2}, ymm2, ymm3 /m256	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed words in ymm3/m256 and ymm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F.WIG 75 /r VPCMPEQW k1 {k2}, zmm2, zmm3 /m512	D	V/V	AVX512BW OR AVX10.1	Compare packed words in zmm3/m512 and zmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare for equality of the packed bytes, words, or doublewords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s.

The (V)PCMPEQB instruction compares the corresponding bytes in the destination and source operands; the (V)PCMPEQW instruction compares the corresponding words in the destination and source operands; and the (V)PCMPEQD instruction compares the corresponding doublewords in the destination and source operands.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX encoded VPCMPEQD: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

EVEX encoded VPCMPEQB/W: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

Operation

PCMPEQB (With 64-bit Operands)

```
    IF DEST[7:0] = SRC[7:0]
        THEN DEST[7:0) := FFH;
        ELSE DEST[7:0] := 0; FI;
    (* Continue comparison of 2nd through 7th bytes in DEST and SRC *)
    IF DEST[63:56] = SRC[63:56]
        THEN DEST[63:56] := FFH;
        ELSE DEST[63:56] := 0; FI;
```

COMPARE_BYTES_EQUAL (SRC1, SRC2)

```
    IF SRC1[7:0] = SRC2[7:0]
    THEN DEST[7:0] := FFH;
    ELSE DEST[7:0] := 0; FI;
(* Continue comparison of 2nd through 15th bytes in SRC1 and SRC2 *)
    IF SRC1[127:120] = SRC2[127:120]
    THEN DEST[127:120] := FFH;
    ELSE DEST[127:120] := 0; FI;
```

COMPARE_WORDS_EQUAL (SRC1, SRC2)

```
    IF SRC1[15:0] = SRC2[15:0]
    THEN DEST[15:0] := FFFFH;
    ELSE DEST[15:0] := 0; FI;
(* Continue comparison of 2nd through 7th 16-bit words in SRC1 and SRC2 *)
    IF SRC1[127:112] = SRC2[127:112]
    THEN DEST[127:112] := FFFFH;
    ELSE DEST[127:112] := 0; FI;
```

COMPARE_DWORDS_EQUAL (SRC1, SRC2)

```
    IF SRC1[31:0] = SRC2[31:0]
    THEN DEST[31:0] := FFFFFFFFH;
    ELSE DEST[31:0] := 0; FI;
(* Continue comparison of 2nd through 3rd 32-bit dwords in SRC1 and SRC2 *)
    IF SRC1[127:96] = SRC2[127:96]
    THEN DEST[127:96] := FFFFFFFFH;
    ELSE DEST[127:96] := 0; FI;
```

PCMPEQB (With 128-bit Operands)

```
DEST[127:0] := COMPARE_BYTES_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPEQB (VEX.128 Encoded Version)

```
DEST[127:0] := COMPARE_BYTES_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAXVL-1:128] := 0
```

VPCMPEQB (VEX.256 Encoded Version)

```
DEST[127:0] := COMPARE_BYTES_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] := COMPARE_BYTES_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] := 0
```

VPCMPEQB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j := 0 TO KL-1
    i := j * 8
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            CMP := SRC1[i+7:i] == SRC2[i+7:i];
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0                    ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

PCMPEQW (With 64-bit Operands)

```
    IF DEST[15:0] = SRC[15:0]
        THEN DEST[15:0] := FFFFH;
        ELSE DEST[15:0] := 0; FI;
    (* Continue comparison of 2nd and 3rd words in DEST and SRC *)
    IF DEST[63:48] = SRC[63:48]
        THEN DEST[63:48] := FFFFH;
        ELSE DEST[63:48] := 0; FI;
```

PCMPEQW (With 128-bit Operands)

```
DEST[127:0] := COMPARE_WORDS_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPEQW (VEX.128 Encoded Version)

```
DEST[127:0] := COMPARE_WORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAXVL-1:128] := 0
```

VPCMPEQW (VEX.256 Encoded Version)

```
DEST[127:0] := COMPARE_WORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] := COMPARE_WORDS_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] := 0
```

VPCMPEQW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            CMP := SRC1[i+15:i] == SRC2[i+15:i];
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0                    ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

PCMPEQD (With 64-bit Operands)

```
    IF DEST[31:0] = SRC[31:0]
        THEN DEST[31:0] := FFFFFFFFH;
        ELSE DEST[31:0] := 0; FI;
    IF DEST[63:32] = SRC[63:32]
        THEN DEST[63:32] := FFFFFFFFH;
        ELSE DEST[63:32] := 0; FI;
```

PCMPEQD (With 128-bit Operands)

```
DEST[127:0] := COMPARE_DWORDS_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPEQD (VEX.128 Encoded Version)

```
DEST[127:0] := COMPARE_DWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAXVL-1:128] := 0
```

VPCMPEQD (VEX.256 Encoded Version)

```
DEST[127:0] := COMPARE_DWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] := COMPARE_DWORDS_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] := 0
```

VPCMPEQD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+31:i] = SRC2[31:0];
                ELSE CMP := SRC1[i+31:i] = SRC2[i+31:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0                    ; zeroing-masking only
    FI;
ENDFOR
```

DEST[MAX_KL-1:KL] := 0

Intel C/C++ Compiler Intrinsic Equivalents

VPCMPEQB __mmask64 _mm512_cmpeq_epi8_mask(__m512i a, __m512i b);
VPCMPEQB __mmask64 _mm512_mask_cmpeq_epi8_mask(__mmask64 k, __m512i a, __m512i b);
VPCMPEQB __mmask32 _mm256_cmpeq_epi8_mask(__m256i a, __m256i b);
VPCMPEQB __mmask32 _mm256_mask_cmpeq_epi8_mask(__mmask32 k, __m256i a, __m256i b);
VPCMPEQB __mmask16 _mm_cmpeq_epi8_mask(__m128i a, __m128i b);
VPCMPEQB __mmask16 _mm_mask_cmpeq_epi8_mask(__mmask16 k, __m128i a, __m128i b);
VPCMPEQW __mmask32 _mm512_cmpeq_epi16_mask(__m512i a, __m512i b);
VPCMPEQW __mmask32 _mm512_mask_cmpeq_epi16_mask(__mmask32 k, __m512i a, __m512i b);
VPCMPEQW __mmask16 _mm256_cmpeq_epi16_mask(__m256i a, __m256i b);
VPCMPEQW __mmask16 _mm256_mask_cmpeq_epi16_mask(__mmask16 k, __m256i a, __m256i b);
VPCMPEQW __mmask8 _mm_cmpeq_epi16_mask(__m128i a, __m128i b);
VPCMPEQW __mmask8 _mm_mask_cmpeq_epi16_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPEQD __mmask16 _mm512_cmpeq_epi32_mask(__m512i a, __m512i b);
VPCMPEQD __mmask16 _mm512_mask_cmpeq_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPEQD __mmask8 _mm256_cmpeq_epi32_mask(__m256i a, __m256i b);
VPCMPEQD __mmask8 _mm256_mask_cmpeq_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPEQD __mmask8 _mm_cmpeq_epi32_mask(__m128i a, __m128i b);
VPCMPEQD __mmask8 _mm_mask_cmpeq_epi32_mask(__mmask8 k, __m128i a, __m128i b);
PCMPEQB __m64 _mm_cmpeq_pi8 (__m64 m1, __m64 m2)
PCMPEQW __m64 _mm_cmpeq_pi16 (__m64 m1, __m64 m2)
PCMPEQD __m64 _mm_cmpeq_pi32 (__m64 m1, __m64 m2)
(V)PCMPEQB __m128i _mm_cmpeq_epi8 (__m128i a, __m128i b)
(V)PCMPEQW __m128i _mm_cmpeq_epi16 (__m128i a, __m128i b)
(V)PCMPEQD __m128i _mm_cmpeq_epi32 (__m128i a, __m128i b)
VPCMPEQB __m256i _mm256_cmpeq_epi8 (__m256i a, __m256i b)
VPCMPEQW __m256i _mm256_cmpeq_epi16 (__m256i a, __m256i b)
VPCMPEQD __m256i _mm256_cmpeq_epi32 (__m256i a, __m256i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded VPCMPEQD, see Table 2-51, "Type E4 Class Exception Conditions."
EVEX-encoded VPCMPEQB/W, see Exceptions Type E4.nb in Table 2-51, "Type E4 Class Exception Conditions."

PCMPEQQ—Compare Packed Qword Data for Equal

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 29 /r PCMPEQQ xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed qwords in xmm2/m128 and xmm1 for equality.
VEX.128.66.0F38.WIG 29 /r VPCMPEQQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed quadwords in xmm3/m128 and xmm2 for equality.
VEX.256.66.0F38.WIG 29 /r VPCMPEQQ ymm1, ymm2, ymm3 /m256	B	V/V	AVX2	Compare packed quadwords in ymm3/m256 and ymm2 for equality.
EVEX.128.66.0F38.W1 29 /r VPCMPEQQ k1 {k2}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare Equal between int64 vector xmm2 and int64 vector xmm3/m128/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F38.W1 29 /r VPCMPEQQ k1 {k2}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare Equal between int64 vector ymm2 and int64 vector ymm3/m256/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F38.W1 29 /r VPCMPEQQ k1 {k2}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Compare Equal between int64 vector zmm2 and int64 vector zmm3/m512/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs an SIMD compare for equality of the packed quadwords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination is set to all 1s; otherwise, it is set to 0s.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX encoded VPCMPEQQ: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

Operation

PCMPEQQ (With 128-bit Operands)

```
IF (DEST[63:0] = SRC[63:0])
    THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
```

```
    ELSE DEST[63:0] := 0; FI;
IF (DEST[127:64] = SRC[127:64])
    THEN DEST[127:64] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] := 0; FI;
DEST[MAXVL-1:128] (Unmodified)
```

COMPARE_QWORDS_EQUAL (SRC1, SRC2)

```
    IF SRC1[63:0] = SRC2[63:0]
    THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] := 0; FI;
    IF SRC1[127:64] = SRC2[127:64]
    THEN DEST[127:64] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] := 0; FI;
```

VPCMPEQQ (VEX.128 Encoded Version)

```
DEST[127:0] := COMPARE_QWORDS_EQUAL(SRC1,SRC2)
DEST[MAXVL-1:128] := 0
```

VPCMPEQQ (VEX.256 Encoded Version)

```
DEST[127:0] := COMPARE_QWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] := COMPARE_QWORDS_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] := 0
```

VPCMPEQQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+63:i] = SRC2[63:0];
                ELSE CMP := SRC1[i+63:i] = SRC2[i+63:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0                    ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCMPEQQ __mmask8 _mm512_cmpeq_epi64_mask(__m512i a, __m512i b);
VPCMPEQQ __mmask8 _mm512_mask_cmpeq_epi64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPEQQ __mmask8 _mm256_cmpeq_epi64_mask(__m256i a, __m256i b);
VPCMPEQQ __mmask8 _mm256_mask_cmpeq_epi64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPEQQ __mmask8 _mm_cmpeq_epi64_mask(__m128i a, __m128i b);
VPCMPEQQ __mmask8 _mm_mask_cmpeq_epi64_mask(__mmask8 k, __m128i a, __m128i b);
(V)PCMPEQQ __m128i _mm_cmpeq_epi64(__m128i a, __m128i b);
VPCMPEQQ __m256i _mm256_cmpeq_epi64(__m256i a, __m256i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded VPCMPEQQ, see Table 2-51, "Type E4 Class Exception Conditions."

PCMPESTRI—Packed Compare Explicit Length Strings, Return Index

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 61 /r imm8 PCMPESTRI xmm1, xmm2/m128, imm8	RMI	V/V	SSE4_2	Perform a packed comparison of string data with explicit lengths, generating an index, and storing the result in ECX.
VEX.128.66.0F3A 61 /r ib VPCMPESTRI xmm1, xmm2/m128, imm8	RMI	V/V	AVX	Perform a packed comparison of string data with explicit lengths, generating an index, and storing the result in ECX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r)	ModRM:r/m (r)	imm8	N/A

Description

The instruction compares and processes data from two string fragments based on the encoded value in the imm8 control byte (see Section 4.1, "Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM"), and generates an index stored to the count register (ECX).

Each string fragment is represented by two values. The first value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). The second value is stored in an input length register. The input length register is EAX/RAX (for xmm1) or EDX/RDX (for xmm2/m128). The length represents the number of bytes/words which are valid for the respective xmm/m128 data.

The length of each input is interpreted as being the absolute-value of the value in the length register. The absolute-value computation saturates to 16 (for bytes) and 8 (for words), based on the value of imm8[bit3] when the value in the length register is greater than 16 (8) or less than -16 (-8).

The comparison and aggregation operations are performed according to the encoded value of imm8 bit fields (see Section 4.1). The index of the first (or last, according to imm8[6]) set bit of IntRes2 (see Section 4.1.4) is returned in ECX. If no bits are set in IntRes2, ECX is set to 16 (8).

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

CFlag – Reset if IntRes2 is equal to zero, set otherwise
ZFlag – Set if absolute-value of EDX is < 16 (8), reset otherwise
SFlag – Set if absolute-value of EAX is < 16 (8), reset otherwise
OFlag – IntRes2[0]
AFlag – Reset
PFlag – Reset

Effective Operand Size

Operating mode/size	Operand 1	Operand 2	Length 1	Length 2	Result
16 bit	xmm	xmm/m128	EAX	EDX	ECX
32 bit	xmm	xmm/m128	EAX	EDX	ECX
64 bit	xmm	xmm/m128	EAX	EDX	ECX
64 bit + REX.W	xmm	xmm/m128	RAX	RDX	ECX

Intel C/C++ Compiler Intrinsic Equivalent For Returning Index

int _mm_cmpestri (__m128i a, int la, __m128i b, int lb, const int mode);

Intel C/C++ Compiler Intrinsics For Reading EFlag Results

int _mm_cmpestra (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestrc (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestrz (__m128i a, int la, __m128i b, int lb, const int mode);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions," additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and:

#UD	If VEX.L = 1.
	If VEX.vvvv ≠ 1111B.

PCMPESTRM—Packed Compare Explicit Length Strings, Return Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 60 /r imm8 PCMPESTRM xmm1, xmm2/m128, imm8	RMI	V/V	SSE4_2	Perform a packed comparison of string data with explicit lengths, generating a mask, and storing the result in XMM0.
VEX.128.66.0F3A 60 /r ib VPCMPESTRM xmm1, xmm2/m128, imm8	RMI	V/V	AVX	Perform a packed comparison of string data with explicit lengths, generating a mask, and storing the result in XMM0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r)	ModRM:r/m (r)	imm8	N/A

Description

The instruction compares data from two string fragments based on the encoded value in the imm8 contol byte (see Section 4.1, "Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM"), and generates a mask stored to XMM0.

Each string fragment is represented by two values. The first value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). The second value is stored in an input length register. The input length register is EAX/RAX (for xmm1) or EDX/RDX (for xmm2/m128). The length represents the number of bytes/words which are valid for the respective xmm/m128 data.

The length of each input is interpreted as being the absolute-value of the value in the length register. The absolute-value computation saturates to 16 (for bytes) and 8 (for words), based on the value of imm8[bit3] when the value in the length register is greater than 16 (8) or less than -16 (-8).

The comparison and aggregation operations are performed according to the encoded value of imm8 bit fields (see Section 4.1). As defined by imm8[6], IntRes2 is then either stored to the least significant bits of XMM0 (zero extended to 128 bits) or expanded into a byte/word-mask and then stored to XMM0.

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

CFlag – Reset if IntRes2 is equal to zero, set otherwise
ZFlag – Set if absolute-value of EDX is < 16 (8), reset otherwise
SFlag – Set if absolute-value of EAX is < 16 (8), reset otherwise
OFlag –IntRes2[0]
AFlag – Reset
PFlag – Reset

Note: In VEX.128 encoded versions, bits (MAXVL-1:128) of XMM0 are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Effective Operand Size

Operating mode/size	Operand 1	Operand 2	Length 1	Length 2	Result
16 bit	xmm	xmm/m128	EAX	EDX	XMM0
32 bit	xmm	xmm/m128	EAX	EDX	XMM0
64 bit	xmm	xmm/m128	EAX	EDX	XMM0
64 bit + REX.W	xmm	xmm/m128	RAX	RDX	XMM0

Intel C/C++ Compiler Intrinsic Equivalent For Returning Mask

__m128i _mm_cmpestrm (__m128i a, int la, __m128i b, int lb, const int mode);

Intel C/C++ Compiler Intrinsics For Reading EFlag Results

int _mm_cmpestra (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestrc (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestrz (__m128i a, int la, __m128i b, int lb, const int mode);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions," additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and:

#UD	If VEX.L = 1.
	If VEX.vvvv ≠ 1111B.

PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Integers for Greater Than

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 64 /r[1] PCMPGTB mm, mm/m64	A	V/V	MMX	Compare packed signed byte integers in mm and mm/m64 for greater than.
66 0F 64 /r PCMPGTB xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed byte integers in xmm1 and xmm2/m128 for greater than.
NP 0F 65 /r[1] PCMPGTW mm, mm/m64	A	V/V	MMX	Compare packed signed word integers in mm and mm/m64 for greater than.
66 0F 65 /r PCMPGTW xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed word integers in xmm1 and xmm2/m128 for greater than.
NP 0F 66 /r[1] PCMPGTD mm, mm/m64	A	V/V	MMX	Compare packed signed doubleword integers in mm and mm/m64 for greater than.
66 0F 66 /r PCMPGTD xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed doubleword integers in xmm1 and xmm2/m128 for greater than.
VEX.128.66.0F.WIG 64 /r VPCMPGTB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 for greater than.
VEX.128.66.0F.WIG 65 /r VPCMPGTW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed word integers in xmm2 and xmm3/m128 for greater than.
VEX.128.66.0F.WIG 66 /r VPCMPGTD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed doubleword integers in xmm2 and xmm3/m128 for greater than.
VEX.256.66.0F.WIG 64 /r VPCMPGTB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed byte integers in ymm2 and ymm3/m256 for greater than.
VEX.256.66.0F.WIG 65 /r VPCMPGTW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed word integers in ymm2 and ymm3/m256 for greater than.
VEX.256.66.0F.WIG 66 /r VPCMPGTD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed doubleword integers in ymm2 and ymm3/m256 for greater than.
EVEX.128.66.0F.W0 66 /r VPCMPGTD k1 {k2}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare Greater between int32 vector xmm2 and int32 vector xmm3/m128/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F.W0 66 /r VPCMPGTD k1 {k2}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare Greater between int32 vector ymm2 and int32 vector ymm3/m256/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F.W0 66 /r VPCMPGTD k1 {k2}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Compare Greater between int32 elements in zmm2 and zmm3/m512/m32bcst, and set destination k1 according to the comparison results under writemask. k2.
EVEX.128.66.0F.WIG 64 /r VPCMPGTB k1 {k2}, xmm2, xmm3/m128	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed byte integers in xmm2 and xmm3/m128 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F.WIG 64 /r VPCMPGTB k1 {k2}, ymm2, ymm3/m256	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed byte integers in ymm2 and ymm3/m256 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F.WIG 64 /r VPCMPGTB k1 {k2}, zmm2, zmm3/m512	D	V/V	AVX512BW OR AVX10.1	Compare packed signed byte integers in zmm2 and zmm3/m512 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.128.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, xmm2, xmm3/m128	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed word integers in xmm2 and xmm3/m128 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, ymm2, ymm3/m256	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed word integers in ymm2 and ymm3/m256 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, zmm2, zmm3/m512	D	V/V	AVX512BW OR AVX10.1	Compare packed signed word integers in zmm2 and zmm3/m512 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs an SIMD signed compare for the greater value of the packed byte, word, or doubleword integers in the destination operand (first operand) and the source operand (second operand). If a data element in the destination operand is greater than the corresponding date element in the source operand, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s.

The PCMPGTB instruction compares the corresponding signed byte integers in the destination and source operands; the PCMPGTW instruction compares the corresponding signed word integers in the destination and source operands; and the PCMPGTD instruction compares the corresponding signed doubleword integers in the destination and source operands.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX encoded VPCMPGTD: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

EVEX encoded VPCMPGTB/W: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

Operation

PCMPGTB (With 64-bit Operands)

```
    IF DEST[7:0] > SRC[7:0]
        THEN DEST[7:0) := FFH;
        ELSE DEST[7:0] := 0; FI;
    (* Continue comparison of 2nd through 7th bytes in DEST and SRC *)
    IF DEST[63:56] > SRC[63:56]
        THEN DEST[63:56] := FFH;
        ELSE DEST[63:56] := 0; FI;
```

COMPARE_BYTES_GREATER (SRC1, SRC2)

```
    IF SRC1[7:0] > SRC2[7:0]
    THEN DEST[7:0] := FFH;
    ELSE DEST[7:0] := 0; FI;
(* Continue comparison of 2nd through 15th bytes in SRC1 and SRC2 *)
    IF SRC1[127:120] > SRC2[127:120]
    THEN DEST[127:120] := FFH;
    ELSE DEST[127:120] := 0; FI;
```

COMPARE_WORDS_GREATER (SRC1, SRC2)

```
    IF SRC1[15:0] > SRC2[15:0]
    THEN DEST[15:0] := FFFFH;
    ELSE DEST[15:0] := 0; FI;
(* Continue comparison of 2nd through 7th 16-bit words in SRC1 and SRC2 *)
    IF SRC1[127:112] > SRC2[127:112]
    THEN DEST[127:112] := FFFFH;
    ELSE DEST[127:112] := 0; FI;
```

COMPARE_DWORDS_GREATER (SRC1, SRC2)

```
    IF SRC1[31:0] > SRC2[31:0]
    THEN DEST[31:0] := FFFFFFFFH;
    ELSE DEST[31:0] := 0; FI;
(* Continue comparison of 2nd through 3rd 32-bit dwords in SRC1 and SRC2 *)
    IF SRC1[127:96] > SRC2[127:96]
    THEN DEST[127:96] := FFFFFFFFH;
    ELSE DEST[127:96] := 0; FI;
```

PCMPGTB (With 128-bit Operands)

```
DEST[127:0] := COMPARE_BYTES_GREATER(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPGTB (VEX.128 Encoded Version)

```
DEST[127:0] := COMPARE_BYTES_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] := 0
```

VPCMPGTB (VEX.256 Encoded Version)

```
DEST[127:0] := COMPARE_BYTES_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] := COMPARE_BYTES_GREATER(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] := 0
```

VPCMPGTB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            CMP := SRC1[i+7:i] > SRC2[i+7:i];
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0                    ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

PCMPGTW (With 64-bit Operands)

```
    IF DEST[15:0] > SRC[15:0]
        THEN DEST[15:0] := FFFFH;
        ELSE DEST[15:0] := 0; FI;
    (* Continue comparison of 2nd and 3rd words in DEST and SRC *)
    IF DEST[63:48] > SRC[63:48]
        THEN DEST[63:48] := FFFFH;
        ELSE DEST[63:48] := 0; FI;
```

PCMPGTW (With 128-bit Operands)

```
DEST[127:0] := COMPARE_WORDS_GREATER(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPGTW (VEX.128 Encoded Version)

```
DEST[127:0] := COMPARE_WORDS_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] := 0
```

VPCMPGTW (VEX.256 Encoded Version)

```
DEST[127:0] := COMPARE_WORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] := COMPARE_WORDS_GREATER(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] := 0
```

VPCMPGTW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            CMP := SRC1[i+15:i] > SRC2[i+15:i];
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0                    ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

PCMPGTD (With 64-bit Operands)

```
    IF DEST[31:0] > SRC[31:0]
        THEN DEST[31:0] := FFFFFFFFH;
        ELSE DEST[31:0] := 0; FI;
    IF DEST[63:32] > SRC[63:32]
        THEN DEST[63:32] := FFFFFFFFH;
        ELSE DEST[63:32] := 0; FI;
```

PCMPGTD (With 128-bit Operands)

```
DEST[127:0] := COMPARE_DWORDS_GREATER(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPGTD (VEX.128 Encoded Version)

```
DEST[127:0] := COMPARE_DWORDS_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] := 0
```

VPCMPGTD (VEX.256 Encoded Version)

```
DEST[127:0] := COMPARE_DWORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] := COMPARE_DWORDS_GREATER(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] := 0
```

VPCMPGTD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+31:i] > SRC2[31:0];
                ELSE CMP := SRC1[i+31:i] > SRC2[i+31:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0                    ; zeroing-masking only
    FI;
ENDFOR
```

```
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPCMPGTB __mmask64 _mm512_cmpgt_epi8_mask(__m512i a, __m512i b);
VPCMPGTB __mmask64 _mm512_mask_cmpgt_epi8_mask(__mmask64 k, __m512i a, __m512i b);
VPCMPGTB __mmask32 _mm256_cmpgt_epi8_mask(__m256i a, __m256i b);
VPCMPGTB __mmask32 _mm256_mask_cmpgt_epi8_mask(__mmask32 k, __m256i a, __m256i b);
VPCMPGTB __mmask16 _mm_cmpgt_epi8_mask(__m128i a, __m128i b);
VPCMPGTB __mmask16 _mm_mask_cmpgt_epi8_mask(__mmask16 k, __m128i a, __m128i b);
VPCMPGTD __mmask16 _mm512_cmpgt_epi32_mask(__m512i a, __m512i b);
VPCMPGTD __mmask16 _mm512_mask_cmpgt_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPGTD __mmask8 _mm256_cmpgt_epi32_mask(__m256i a, __m256i b);
VPCMPGTD __mmask8 _mm256_mask_cmpgt_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPGTD __mmask8 _mm_cmpgt_epi32_mask(__m128i a, __m128i b);
VPCMPGTD __mmask8 _mm_mask_cmpgt_epi32_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPGTW __mmask32 _mm512_cmpgt_epi16_mask(__m512i a, __m512i b);
VPCMPGTW __mmask32 _mm512_mask_cmpgt_epi16_mask(__mmask32 k, __m512i a, __m512i b);
VPCMPGTW __mmask16 _mm256_cmpgt_epi16_mask(__m256i a, __m256i b);
VPCMPGTW __mmask16 _mm256_mask_cmpgt_epi16_mask(__mmask16 k, __m256i a, __m256i b);
VPCMPGTW __mmask8 _mm_cmpgt_epi16_mask(__m128i a, __m128i b);
VPCMPGTW __mmask8 _mm_mask_cmpgt_epi16_mask(__mmask8 k, __m128i a, __m128i b);
PCMPGTB __m64 _mm_cmpgt_pi8 (__m64 m1, __m64 m2)
PCMPGTW __m64 _mm_cmpgt_pi16 (__m64 m1, __m64 m2)
PCMPGTD __m64 _mm_cmpgt_pi32 (__m64 m1, __m64 m2)
(V)PCMPGTB __m128i _mm_cmpgt_epi8 (__m128i a, __m128i b)
(V)PCMPGTW __m128i _mm_cmpgt_epi16 (__m128i a, __m128i b)
(V)DCMPGTD __m128i _mm_cmpgt_epi32 (__m128i a, __m128i b)
VPCMPGTB __m256i _mm256_cmpgt_epi8 (__m256i a, __m256i b)
VPCMPGTW __m256i _mm256_cmpgt_epi16 (__m256i a, __m256i b)
VPCMPGTD __m256i _mm256_cmpgt_epi32 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded VPCMPGTD, see Table 2-51, "Type E4 Class Exception Conditions."
EVEX-encoded VPCMPGTB/W, see Exceptions Type E4.nb in Table 2-51, "Type E4 Class Exception Conditions."

PCMPGTQ—Compare Packed Data for Greater Than

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 37 /r PCMPGTQ xmm1,xmm2/m128	A	V/V	SSE4_2	Compare packed signed qwords in xmm2/m128 and xmm1 for greater than.
VEX.128.66.0F38.WIG 37 /r VPCMPGTQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed qwords in xmm2 and xmm3/m128 for greater than.
VEX.256.66.0F38.WIG 37 /r VPCMPGTQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed qwords in ymm2 and ymm3/m256 for greater than.
EVEX.128.66.0F38.W1 37 /r VPCMPGTQ k1 {k2}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare Greater between int64 vector xmm2 and int64 vector xmm3/m128/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F38.W1 37 /r VPCMPGTQ k1 {k2}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare Greater between int64 vector ymm2 and int64 vector ymm3/m256/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F38.W1 37 /r VPCMPGTQ k1 {k2}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Compare Greater between int64 vector zmm2 and int64 vector zmm3/m512/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs an SIMD signed compare for the packed quadwords in the destination operand (first operand) and the source operand (second operand). If the data element in the first (destination) operand is greater than the corresponding element in the second (source) operand, the corresponding data element in the destination is set to all 1s; otherwise, it is set to 0s.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX encoded VPCMPGTD/Q: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

Operation

COMPARE_QWORDS_GREATER (SRC1, SRC2)

```
    IF SRC1[63:0] > SRC2[63:0]
    THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] := 0; FI;
    IF SRC1[127:64] > SRC2[127:64]
    THEN DEST[127:64] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] := 0; FI;
```

VPCMPGTQ (VEX.128 Encoded Version)

```
DEST[127:0] := COMPARE_QWORDS_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] := 0
```

VPCMPGTQ (VEX.256 Encoded Version)

```
DEST[127:0] := COMPARE_QWORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] := COMPARE_QWORDS_GREATER(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] := 0
```

VPCMPGTQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+63:i] > SRC2[63:0];
                ELSE CMP := SRC1[i+63:i] > SRC2[i+63:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0                    ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCMPGTQ __mmask8 _mm512_cmpgt_epi64_mask(__m512i a, __m512i b);
VPCMPGTQ __mmask8 _mm512_mask_cmpgt_epi64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPGTQ __mmask8 _mm256_cmpgt_epi64_mask(__m256i a, __m256i b);
VPCMPGTQ __mmask8 _mm256_mask_cmpgt_epi64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPGTQ __mmask8 _mm_cmpgt_epi64_mask(__m128i a, __m128i b);
VPCMPGTQ __mmask8 _mm_mask_cmpgt_epi64_mask(__mmask8 k, __m128i a, __m128i b);
(V)PCMPGTQ __m128i _mm_cmpgt_epi64(__m128i a, __m128i b)
VPCMPGTQ __m256i _mm256_cmpgt_epi64(__m256i a, __m256i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded VPCMPGTQ, see Table 2-51, “Type E4 Class Exception Conditions.”

PCMPISTRI—Packed Compare Implicit Length Strings, Return Index

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 63 /r imm8 PCMPISTRI xmm1, xmm2/m128, imm8	RM	V/V	SSE4_2	Perform a packed comparison of string data with implicit lengths, generating an index, and storing the result in ECX.
VEX.128.66.0F3A.WIG 63 /r ib VPCMPISTRI xmm1, xmm2/m128, imm8	RM	V/V	AVX	Perform a packed comparison of string data with implicit lengths, generating an index, and storing the result in ECX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	imm8	N/A

Description

The instruction compares data from two strings based on the encoded value in the imm8 control byte (see Section 4.1, "Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM"), and generates an index stored to ECX.

Each string is represented by a single value. The value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). Each input byte/word is augmented with a valid/invalid tag. A byte/word is considered valid only if it has a lower index than the least significant null byte/word. (The least significant null byte/word is also considered invalid.)

The comparison and aggregation operations are performed according to the encoded value of imm8 bit fields (see Section 4.1). The index of the first (or last, according to imm8[6]) set bit of IntRes2 is returned in ECX. If no bits are set in IntRes2, ECX is set to 16 (8).

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

CFlag – Reset if IntRes2 is equal to zero, set otherwise
ZFlag – Set if any byte/word of xmm2/mem128 is null, reset otherwise
SFlag – Set if any byte/word of xmm1 is null, reset otherwise
OFlag –IntRes2[0]
AFlag – Reset
PFlag – Reset

Note: In VEX.128 encoded version, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Effective Operand Size

Operating mode/size	Operand 1	Operand 2	Result
16 bit	xmm	xmm/m128	ECX
32 bit	xmm	xmm/m128	ECX
64 bit	xmm	xmm/m128	ECX

Intel C/C++ Compiler Intrinsic Equivalent For Returning Index

int _mm_cmpistri (__m128i a, __m128i b, const int mode);

Intel C/C++ Compiler Intrinsics For Reading EFlag Results

int _mm_cmpistra (__m128i a, __m128i b, const int mode);
int _mm_cmpistrc (__m128i a, __m128i b, const int mode);
int _mm_cmpistro (__m128i a, __m128i b, const int mode);
int _mm_cmpistrs (__m128i a, __m128i b, const int mode);
int _mm_cmpistrz (__m128i a, __m128i b, const int mode);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions," additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and:

#UD	If VEX.L = 1.
	If VEX.vvvv ≠ 1111B.

PCMPISTRM—Packed Compare Implicit Length Strings, Return Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 62 /r imm8 PCMPISTRM xmm1, xmm2/m128, imm8	RM	V/V	SSE4_2	Perform a packed comparison of string data with implicit lengths, generating a mask, and storing the result in XMM0.
VEX.128.66.0F3A.WIG 62 /r ib VPCMPISTRM xmm1, xmm2/m128, imm8	RM	V/V	AVX	Perform a packed comparison of string data with implicit lengths, generating a Mask, and storing the result in XMM0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	imm8	N/A

Description

The instruction compares data from two strings based on the encoded value in the imm8 byte (see Section 4.1, "Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM") generating a mask stored to XMM0.

Each string is represented by a single value. The value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). Each input byte/word is augmented with a valid/invalid tag. A byte/word is considered valid only if it has a lower index than the least significant null byte/word. (The least significant null byte/word is also considered invalid.)

The comparison and aggregation operation are performed according to the encoded value of imm8 bit fields (see Section 4.1). As defined by imm8[6], IntRes2 is then either stored to the least significant bits of XMM0 (zero extended to 128 bits) or expanded into a byte/word-mask and then stored to XMM0.

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

CFlag – Reset if IntRes2 is equal to zero, set otherwise
ZFlag – Set if any byte/word of xmm2/mem128 is null, reset otherwise
SFlag – Set if any byte/word of xmm1 is null, reset otherwise
OFlag – IntRes2[0]
AFlag – Reset
PFlag – Reset

Note: In VEX.128 encoded versions, bits (MAXVL-1:128) of XMM0 are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Effective Operand Size

Operating mode/size	Operand 1	Operand 2	Result
16 bit	xmm	xmm/m128	XMM0
32 bit	xmm	xmm/m128	XMM0
64 bit	xmm	xmm/m128	XMM0

Intel C/C++ Compiler Intrinsic Equivalent For Returning Mask

__m128i _mm_cmpistrm (__m128i a, __m128i b, const int mode);

Intel C/C++ Compiler Intrinsics For Reading EFlag Results

int _mm_cmpistra (__m128i a, __m128i b, const int mode);
int _mm_cmpistrc (__m128i a, __m128i b, const int mode);
int _mm_cmpistro (__m128i a, __m128i b, const int mode);
int _mm_cmpistrs (__m128i a, __m128i b, const int mode);
int _mm_cmpistrz (__m128i a, __m128i b, const int mode);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions," additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and:

#UD	If VEX.L = 1.
	If VEX.vvvv ≠ 1111B.

PCONFIG—Platform Configuration

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 C5 PCONFIG	A	V/V	PCONFIG	This instruction is used to execute functions for configuring platform features.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	N/A	N/A	N/A	N/A

Description

The PCONFIG instruction allows software to configure certain platform features. It supports these features with multiple leaf functions, selecting a leaf function using the value in EAX.

Depending on the leaf function, the registers RBX, RCX, and RDX may be used to provide input information or for the instruction to report output information. Addresses and operands are 32 bits outside 64-bit mode and are 64 bits in 64-bit mode. The value of CS.D does not affect operand size or address size.

Executions of PCONFIG may fail for platform-specific reasons. An execution reports failure by setting the ZF flag and loading EAX with a non-zero failure reason; a successful execution clears ZF and EAX.

Each PCONFIG leaf function applies to a specific hardware block called a PCONFIG target. The leaf function is supported only if the processor supports that target. Each target is associated with a numerical target identifier, and CPUID leaf 1BH (PCONFIG information) enumerates the identifiers of the supported targets. An attempt to execute an undefined leaf function, or a leaf function that applies to an unsupported target identifier, results in a general-protection exception (#GP).

Leaf Function MKTME_KEY_PROGRAM

PCONFIG leaf function 0 (selected by loading EAX with value 0) is used for key programming for total memory encryption-multi-key (TME-MK).[1] This leaf function is called MKTME_KEY_PROGRAM and it pertains to the TME-MK target, which has target identifier 1. The leaf function uses the EBX (or RBX) register for additional input information.

Software uses this leaf function to manage the encryption key associated with a particular key identifier (KeyID). The leaf function uses a data structure called the **TME-MK key programming structure** (MKTME_KEY_PROGRAM_STRUCT). Software provides the address of the structure (as an offset in the DS segment) in EBX (or RBX). The format of the structure is given in Table 4-16.

Table 4-16. MKTME_KEY_PROGRAM_STRUCT Format

Field	Offset (bytes)	Size (bytes)	Comments
KEYID	0	2	Key Identifier.
KEYID_CTRL	2	4	KeyID control: ▪ Bits 7:0: key-programming command (COMMAND) ▪ Bits 23:8: encryption algorithm (ENC_ALG) ▪ Bits 31:24: Reserved, must be zero (RSVD)
Ignored	6	58	Not used.
KEY_FIELD_1	64	64	Software supplied data key or entropy for data key.
KEY_FIELD_2	128	64	Software supplied tweak key or entropy for tweak key.

1. Further details on TME-MK can be found here: https://software.intel.com/sites/default/files/managed/a5/16/Multi-Key-Total-Memory-Encryption-Spec.pdf

A description of each of the fields in MKTME_KEY_PROGRAM_STRUCT is provided below:

- **KEYID:** The key identifier (KeyID) being programmed to the TME-MK engine. This leaf function causes a general-protection exception (#GP) if the KeyID is zero. KeyID zero always uses the current behavior configured for TME (total memory encryption), either to encrypt with platform TME key or to bypass TME encryption. This leaf function also causes a #GP if the KeyID exceeds the maximum enumerated IA32_TME_CAPABILITY.MK_TME_MAX_KEYS[50:36] or configured by the setting of IA32_TME_ACTIVATE.MK_TME_KEYID_BITS[39:32].
  - If IA32_TME_ACTIVATE.MK_TME_KEYID_BITS[35:32] = *k*, KEYID[15:*k*] must be zero.
  - If the logical processor is outside secure arbitration mode (SEAM; see Chapter 35) and IA32_TME_ACTIVATE.TDX_RESERVED_KEYID_BITS[39:36] = *p* > 0, KEYID[*k*–1:*k*–*p*] must also be zero. (Note that *p* will never exceed *k*.)

  If either is not true, the leaf function causes #GP.
- **KEYID_CTRL:** The KEYID_CTRL field comprises two sub-fields used by software to control the encryption performed for the selected KeyID:
  - Key-programming command (COMMAND; bits 7:0). This 8-bit field should contain one of the following values:
    - KEYID_SET_KEY_DIRECT (value 0). With this command, software programs directly the encryption key to be used for the selected KeyID.
    - KEYID_SET_KEY_RANDOM (value 1). With this command, software has the CPU generate and assign an encryption key to be used for the selected KeyID using a hardware random-number generator.

      If this command is used and there is insufficient entropy for the random-number generator, the leaf function will fail and report the failure by loading EAX with value 2 (ENTROPY_ERROR).

      Because the keys programed by this leaf function are discarded on reset and software cannot read the programmed keys, the keys programmed with this command are ephemeral.
    - KEYID_CLEAR_KEY (value 2). With this command, software indicates that the selected KeyID should use the current behavior configured for TME (see above).
    - KEYID_NO_ENCRYPT (value 3). With this command, software indicates that no encryption should be used for the selected KeyID.

    If any other value is used, the leaf function causes a #GP.
  - Encryption algorithm (ENC_ALG, bits 23:8). Bits 63:48 of the IA32_TME_ACTIVATE MSR (MSR index 982H) indicate which encryption algorithms are supported by the platform. The 16-bit ENC_ALG field should specify one of the algorithms indicated in IA32_TME_ACTIVATE. This leaf function causes a #GP if ENC_ALG does not set exactly one bit or if it sets a bit whose corresponding bit is not set in IA32_TME_ACTIVATE[63:48].
- **KEY_FIELD_1:** Use of this field depends upon selected key-programming command:
  - If the direct key-programming command is used (KEYID_SET_KEY_DIRECT), this field carries the software supplied data key to be used for the KeyID.
  - If the random key-programming command is used (KEYID_SET_KEY_RANDOM), this field carries the software supplied entropy to be mixed in the CPU generated random data key.
  - This field is ignored when one of the other key-programming commands is used.

  It is software's responsibility to ensure that the key supplied for the direct key-programming option or the entropy supplied for the random key-programming option does not result in weak keys. There are no explicit checks in the instruction to detect or prevent weak keys.
- **KEY_FIELD_2:** Use of this field depends upon selected key-programming command:
  - If the direct key-programming command is used (KEYID_SET_KEY_DIRECT), this field carries the software supplied tweak key to be used for the KeyID.
  - If the random key-programming command is used (KEYID_SET_KEY_RANDOM), this field carries the software supplied entropy to be mixed in the CPU generated random tweak key.
  - This field is ignored when one of the other key-programming commands is used.

It is software's responsibility to ensure that the key supplied for the direct key-programming option or the entropy supplied for the random key-programming option does not result in weak keys. There are no explicit checks in the instruction to detect or prevent weak keys.

All KeyIDs default to TME behavior (encrypt with TME key or bypass encryption) on activation of TME-MK. Software can at any point decide to change the key for a KeyID using this leaf function. Changing the key for a KeyID does **not** change the state of the TLB caches or memory pipeline. Software is responsible for taking appropriate actions to ensure correct behavior.

The key table used by TME-MK is shared by all logical processors in a platform. For this reason, execution of the MKTME_KEY_PROGRAM leaf function must gain exclusive access to the key table before updating it. The leaf function does this by acquiring lock (implemented in the platform) and retaining that lock until the execution completes. An execution of the leaf function may fail to acquire the lock if it is already in use. In this situation, the leaf function will load EAX with failure reason 5 (DEVICE_BUSY) indicating that software must retry. When this happens, the key table is not updated, and software should retry execution of PCONFIG.

Leaf Function TSE_KEY_PROGRAM

PCONFIG leaf function 1(selected by loading EAX with value 1) is used for direct key programming for total storage encryption (TSE). This leaf function is called TSE_KEY_PROGRAM and it pertains to the TSE target, which has target identifier 2. The leaf function can be used only in 64-bit mode. It uses the RBX register for additional input information.

Software uses this leaf function to manage the encryption key associated with a particular key identifier (KeyID). The leaf function uses a data structure called the **TSE key programming structure** (TSE_KEY_PROGRAM_-STRUCT). Software provides the linear address of the structure in RBX. The format of the structure is given in Table 4-17.

Table 4-17. TSE_KEY_PROGRAM_STRUCT Format

Field	Offset (bytes)	Size (bytes)	Comments
KEYID	0	2	Key Identifier.
KEYID_CTRL	2	4	KeyID control: ▪ Bits 7:0: key-programming command (COMMAND) ▪ Bits 23:8: encryption algorithm (ENC_ALG) ▪ Bits 31:24: Reserved, must be zero (RSVD)
Ignored	6	58	Not used.
KEY_FIELD_1	64	64	Software supplied data key.
KEY_FIELD_2	128	64	Software supplied tweak key.

A description of each of the fields in MKTME_KEY_PROGRAM_STRUCT is provided below:

- **KEYID:** The key identifier (KeyID) being programmed to the TSE engine. The leaf function causes a general-protection exception (#GP) if the KeyID exceeds the maximum enumerated in the TSE_MAX_KEYS field (bits 50:36) of the IA32_TSE_CAPABILITY MSR (MSR index 9F1H).
- **KEYID_CTRL:** The KEYID_CTRL field comprises two sub-fields used by software to control the encryption performed for the selected KeyID:
  - Key-programming command (COMMAND; bits 7:0). This 8-bit field should contain one of the following values:
    - TSE_SET_KEY_DIRECT (value 0). With this command, software programs directly the encryption key to be used for the selected KeyID.
    - TSE_NO_ENCRYPT (value 1). With this command, software indicates that no encryption should be used for the selected KeyID.

    If any other value is used, the leaf function causes a #GP.
  - Encryption algorithm (ENC_ALG, bits 23:8). IA32_TSE_CAPABILITY[15:0] indicates which encryption algorithms are supported by the platform. The 16-bit ENC_ALG field should specify one of the algorithms

indicated in IA32_TSE_CAPABILITY. The leaf function causes a #GP if ENC_ALG does not set exactly one bit or if it sets a bit whose corresponding bit is not set in IA32_TSE_CAPABILITY.

- **KEY_FIELD_1:** If the direct key-programming command is used (TSE_SET_KEY_DIRECT), this field carries the software supplied data key to be used for the KeyID. Otherwise, the field is ignored.
- **KEY_FIELD_2:** If the direct key-programming command is used (TSE_SET_KEY_DIRECT), this field carries the software supplied tweak key to be used for the KeyID. Otherwise, the field is ignored.

The TSE key table is shared by all logical processors in a platform. For this reason, execution of this leaf function must gain exclusive access to the key table before updating it. The leaf function does this by acquiring a lock (implemented in the platform) and retaining that lock until the execution completes. An execution of the leaf function may fail to acquire the lock if it is already in use. In this situation, the leaf function will load EAX with failure reason 5 (DEVICE_BUSY). When this happens, the key table is not updated, and software should retry execution of PCONFIG.

NOTES

Earlier versions of this manual specified that bytes 63:6 of MKTME_KEY_PROGRAM_STRUCT were reserved and that leaf function MKTME_PROGRAM would cause a #GP if they were not all zero. This is not the case. As indicated in Table 4-16, PCONFIG ignores those bytes.

They also specified that leaf function would cause a #GP if the upper 48 bytes of each of the 64-byte key fields were not all 0. This is not the case. From each of these fields, the leaf function uses the number of bytes required by the selected encryption algorithm (e.g., 32 bytes for AES-XTS 256) and ignores the upper bytes.

They also specified that the leaf function would complete and report a failure reason in EAX if the structure specified an incorrect KeyID, and unsupported key-programming command, or an incorrect selection of an encryption algorithm. This is not the case. As indicated above (and in the Operation section), those conditions cause #GP.

Leaf Function TSE_KEY_PROGRAM_WRAPPED

PCONFIG leaf function 2 (selected by loading EAX with value 2) is used for wrapped key programming for total storage encryption (TSE). This leaf function is called TSE_KEY_PROGRAM_WRAPPED and it pertains to the TSE target, which has target identifier 2. The leaf function can be used only in 64-bit mode. It uses the RBX and RCX registers for additional input information.

Software uses this leaf function to manage the encryption key associated with a particular key identifier (KeyID). The leaf function uses control input provided in RBX. The format of that input is given in Table 4-18.

Table 4-18. TSE_KEY_PROGRAM_WRAPPED Control Input

Field	Bit Positions	Comments
KEYID	15:0	Key identifier.
Reserved	23:16	Reserved, must be zero.
ENC_ALG	39:24	Encryption algorithm.
Ignored	63:40	Not used.

A description of each of the fields in the control input is provided below:

- **KEYID:** The key identifier (KeyID) being programmed to the TSE engine. The leaf function causes a general-protection exception (#GP) if the KeyID exceeds the maximum enumerated in the TSE_MAX_KEYs field (bits 50:36) of the IA32_TSE_CAPABILITY MSR (MSR index 9F1H).
- **ENC_ALG:** The encryption algorithm selected for the KeyID. IA32_TSE_CAPABILITY[15:0] indicates which encryption algorithms are supported by the platform. The 16-bit ENC_ALG field should specify one of the algorithms indicated in IA32_TSE_CAPABILITY. The leaf function causes a #GP if ENC_ALG does not set exactly one bit or if it sets a bit whose corresponding bit is not set in IA32_TSE_CAPABILITY.

The leaf function also uses a 256-byte data structure called the **bind structure**. This structure should be the output of the PBNDKB instruction, subsequently modified by software (see below). Software provides the linear address of the structure in RCX. The format of the structure is given in Table 4-19.

Table 4-19. Bind Structure Format

Field	Offset (bytes)	Size (bytes)	Comments
MAC	0	16	MAC produced by PBNDKB of its input bind structure
Reserved	16	8	Reserved, must be zero.
IV	24	12	Initialization vector.
Reserved	36	28	Reserved, must be zero.
BTENCDATA	64	64	Encrypted data (data key and tweak key)
BTDATA	128	128	Additional control and data (not encrypted)

A description of each of the fields in TSE_BIND_STRUCT is provided below:

- **MAC:** A MAC produced by PBNDKB of its input bind structure. The PCONFIG leaf function will recompute the MAC and confirm that it matches this value.
- **IV:** The initialization vector that PBNDKB used for encryption. The PCONFIG leaf function will use this in its decryption of encrypted data and computation of the MAC.
- **BTENCDATA:** Data which had been encrypted by PBNDKB, containing the data and tweak keys to be used by TSE.
- **BTDATA:** Data that was input to PBNDKB that was output without encryption. It has the following format:
  - **USER_SUPP_CHALLENGE** (bytes 31:0): PBNDKB uses a value provided by software in its input bind structure but writes zero to this field in the output bind structure to be used by PCONFIG. Software should configure this field with the proper value before executing this PCONFIG leaf function.
  - **KEY_GENERATION_CTRL** (byte 32): PBNDKB uses this value to determine whether to generate random keys. The PCONFIG leaf function does not use this field.
  - The remaining 95 bytes are reserved and must be zero.

  The leaf function uses the entire BTDATA field when it computes the MAC.

The leaf function determines a 256-bit **wrapping key** by computing an HMAC based on SHA-256 using 256-bit platform-specific key and the USER_SUPP_CHALLENGE in the BTDATA field of the TSE_BIND_STRUCT.

Using the wrapping key, the leaf function uses an AES GCM authenticated decryption function to decrypt BTENC-DATA and compute a MAC. The decryption function uses the following inputs:

- The 64-byte BTENCDATA from TSE_BIND_STRUCT to be decrypted.
- The 256-bit wrapping key.
- The 96-bit IV from TSE_BIND_STRUCT.
- Additional authenticated data that is the concatenation of bytes 63:16 and bytes 255:128 of the TSE_BIND_-STRUCT. These 176 bytes will comprise 8 bytes of zeroes, the 12-byte IV, 28 bytes of zeroes, and 128 bytes of BTDATA of which the upper 95 bytes are zero).
- The length of the additional authenticated data (176).

The decryption function produces a structure with a 64 bytes of decrypted data and a 16-byte MAC. The decrypted data comprises a 256-bit data key and a 256-bit tweak key.

If the MAC produced by the decryption function differs from that provided in the TSE_BIND_STRUCT, the leaf func-tion will load EAX with failure reason 7 (UNWRAP_FAILURE). Otherwise, the leaf function will attempt to program the TSE key table for the selected KeyID with the keys contained in the decrypted data.

The TSE key table is shared by all logical processors in a platform. For this reason, execution of this leaf function must gain exclusive access to the key table before updating it. The leaf function does this by acquiring a lock (implemented in the platform) and retaining that lock until the execution completes. An execution of the leaf func-tion may fail to acquire the lock if it is already in use. In this situation, the leaf function will load EAX with failure reason 5 (DEVICE_BUSY). When this happens, the key table is not updated, and software should retry execution of PCONFIG.

Operation

```
(* #UD if PCONFIG is not enumerated or CPL > 0 *)
IF CPUID.07H.00H:EDX.PCONFIG[18]= 0 OR CPL > 0
    THEN #UD; FI;

(* #GP(0) for an unsupported leaf function *)
IF EAX > 2
    THEN #GP(0); FI;

CASE (EAX)          (* operation based on selected leaf function *)
    0 (MKTME_KEY_PROGRAM):
    IF CPUID function 1BH does not enumerate support for the TME-MK target (value 1)
        THEN #GP(0); FI;
    (* Confirm that TME-MK is properly enabled by the IA32_TME_ACTIVATE MSR *)
    (* The MSR must be locked, encryption enabled, and a non-zero number of KeyID bits specified *)
    IF IA32_TME_ACTIVATE[0] = 0 OR IA32_TME_ACTIVATE[1] = 0 OR IA32_TME_ACTIVATE[35:32] = 0
            THEN #GP(0); FI;

    IF DS:RBX is not 256-byte aligned
        THEN #GP(0); FI;

    Load TMP_KEY_PROGRAM_STRUCT from 192 bytes at linear address DS:RBX;

    IF TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL sets any reserved bits
        THEN #GP(0); FI;

    (* Check for a valid command *)
    IF TMP_KEY_PROGRAM_STRUCT. KEYID_CTRL.COMMAND > 3
        THEN #GP(0); FI;

    (* Check that the KEYID being operated upon is a valid KEYID *)
    IF TMP_KEY_PROGRAM_STRUCT.KEYID = 0 OR
        TMP_KEY_PROGRAM_STRUCT.KEYID > IA32_TME_CAPABILITY.MK_TME_MAX_KEYS
            THEN #GP(0); FI;
    k := IA32_TME_ACTIVATE.MK_TME_KEYID_BITS;
    IF TMP_KEY_PROGRAM_STRUCT.KEYID[15:k] != 0
        THEN #GP(0); FI;
    IF not in SEAM AND IA32_TME_ACTIVATE.TDX_RESERVED_KEYID_BITS > 0
        THEN
            p := IA32_TME_ACTIVATE.TDX_RESERVED_KEYID_BITS;
            IF TMP_KEY_PROGRAM_STRUCT.KEYID[k–1:k–p] != 0
                THEN #GP(0); FI;
    FI;

    (* Check that only one encryption algorithm is requested for the KeyID and it is one of the activated algorithms *)
    IF TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.ENC_ALG does not set exactly one bit OR
        (TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.ENC_ALG & IA32_TME_ACTIVATE[63:48]) = 0
            THEN #GP(0); FI:

    Attempt to acquire lock to gain exclusive access to platform key table for TME-MK;
    IF attempt is unsuccessful
        THEN (* PCONFIG failure *)
            RFLAGS.ZF := 1;
            RAX := DEVICE_BUSY;   (* failure reason 5 *)
```

```
            GOTO EXIT;
FI;

CASE (TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.COMMAND) OF
    0 (KEYID_SET_KEY_DIRECT):
    Update TME-MK table for TMP_KEY_PROGRAM_STRUCT.KEYID as follows:
        Encrypt with the selected key
        Use the encryption algorithm selected by TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.ENC_ALG
        (* The number of bytes used by the next two lines depends on selected encryption algorithm *)
        DATA_KEY is TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_1
        TWEAK_KEY is TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_2
    BREAK;

    1 (KEYID_SET_KEY_RANDOM):
    Load TMP_RND_DATA_KEY with a random key using hardware RNG; (* key size depends on selected encryption algorithm *)
    IF there was insufficient entropy
        THEN (* PCONFIG failure *)
            RFLAGS.ZF := 1;
            RAX := ENTROPY_ERROR;    (* failure reason 2 *)
            Release lock on platform key table;
            GOTO EXIT;
    FI;
    Load TMP_RND_TWEAK_KEY with a random key using hardware RNG; (* key size depends on selected encryption algorithm *)
    IF there was insufficient entropy
        THEN (* PCONFIG failure *)
            RFLAGS.ZF := 1;
            RAX := ENTROPY_ERROR;    (* failure reason 2 *)
            Release lock on platform key table;
            GOTO EXIT;
    FI;
    (* Combine software-supplied entropy to the data key and tweak key *)
    (* The number of bytes used by the next two lines depends on selected encryption algorithm *)
    TMP_RND_DATA_KEY := TMP_RND_KEY XOR TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_1;
    TMP_RND_TWEAK_KEY := TMP_RND_TWEAK_KEY XOR TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_2;

    Update TME-MK table for TMP_KEY_PROGRAM_STRUCT.KEYID as follows:
        Encrypt with the selected key
        Use the encryption algorithm selected by TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.ENC_ALG
        (* The number of bytes used by the next two lines depends on selected encryption algorithm *)
        DATA_KEY is TMP_RND_DATA_KEY
        TWEAK_KEY is TMP_RND_TWEAK_KEY
    BREAK;

    2 (KEYID_CLEAR_KEY):
    Update TME-MK table for TMP_KEY_PROGRAM_STRUCT.KEYID as follows:
        Encrypt (or not) using the current configuration for TME
        The specified encryption algorithm and key values are not used.
    BREAK;

    3 (KEYID_NO_ENCRYPT):
    Update TME-MK table for TMP_KEY_PROGRAM_STRUCT.KEYID as follows:
        Do not encrypt
        The specified encryption algorithm and key values are not used.
    BREAK;
```

```
ESAC;
Release lock on platform key table for TME-MK;

1 (TSE_KEY_PROGRAM):
IF CPUID function 1BH does not enumerate support for the TSE target (value 2)
      THEN #GP(0); FI;

IF not in 64-bit mode
      THEN #GP(0); FI;

IF RBX is not 256-byte aligned
      THEN #GP(0); FI;

Load TMP_KEY_STRUCT from 192 bytes at linear address in RBX;

IF TMP_KEY_STRUCT.KEYID_CTRL sets any reserved bits
      THEN #GP(0); FI;

(* Check for a valid command *)
IF TMP_KEY_STRUCT. KEYID_CTRL.COMMAND > 1
      THEN #GP(0); FI;

(* Check that the KEYID being operated upon is a valid KEYID *)
IF TMP_KEY_STRUCT.KEYID > IA32_TSE_CAPABILITY.TSE_MAX_KEYS
      THEN #GP(0); FI;

(* Check that only one encryption algorithm is requested for the KeyID and it is one of the activated algorithms *)
IF TMP_KEY_STRUCT.KEYID_CTRL.ENC_ALG does not set exactly one bit OR
      (TMP_KEY_STRUCT.KEYID_CTRL.ENC_ALG & IA32_TSE_CAPABILITY[15:0]) = 0
            THEN #GP(0); FI;

Attempt to acquire lock to gain exclusive access to platform key table for TSE;
IF attempt is unsuccessful
      THEN (* PCONFIG failure *)
            RFLAGS.ZF := 1;
            RAX := DEVICE_BUSY;   (* failure reason 5 *)
            GOTO EXIT;
FI;

CASE (TMP_KEY_STRUCT.KEYID_CTRL.COMMAND) OF
      0 (TSE_SET_KEY_DIRECT):
      Update TSE table for TMP_KEY_STRUCT.KEYID as follows:
            Encrypt with the selected key
            Use the encryption algorithm selected by TMP_KEY_STRUCT.KEYID_CTRL.ENC_ALG
            (* The number of bytes used by the next two lines depends on selected encryption algorithm *)
            DATA_KEY is TMP_KEY_STRUCT.KEY_FIELD_1
            TWEAK_KEY is TMP_KEY_STRUCT.KEY_FIELD_2
      BREAK;

      1 (TSE_NO_ENCRYPT):
      Update TSE table for TMP_KEY_STRUCT.KEYID as follows:
            Do not encrypt
            The specified encryption algorithm and key values are not used.
      BREAK;
```

```
ESAC;
Release lock on platform key table for TSE;

2 (TSE_KEY_PROGRAM_WRAPPED):
IF CPUID function 1BH does not enumerate support for the TSE target (value 2)
      THEN #GP(0); FI;

IF not in 64-bit mode OR RBX[23:16] != 0 OR RCX is not 256-byte aligned
      THEN #GP(0); FI;

(* Check that the KEYID being operated upon is a valid KEYID *)
IF RBX[15:0] > IA32_TSE_CAPABILITY.TSE_MAX_KEYS
      THEN #GP(0); FI;

(* Check that only one encryption algorithm is requested for the KeyID and it is one of the activated algorithms *)
IF RBX[39:24] does not set exactly one bit OR (RBX[39:24] & IA32_TSE_CAPABILITY[15:0]) = 0
      THEN #GP(0); FI;

Load TMP_BIND_STRUCT from 256 bytes at linear address in RCX;

(* Check TMP_BIND_STRUCT for illegal values *)
IF bytes 23:16 and bytes 63:36 of TMP_BIND_STRUCT are not all zero
      THEN #GP(0); FI;
IF TMP_BIND_STRUCT.BTDATA.KEY_GENERATION_CTRL > 1
      THEN #GP(0); FI;
IF bytes 128:33 of TMP_BIND_STRUCT.BTDATA are not all zero
      THEN #GP(0); FI;

(* Compute wrapping key *)
PLATFORM_KEY := 256-bit platform-specific key;
WRAPPING_KEY := HMAC_SHA256(PLATFORM_KEY, TMP_BIND_STRUCT.BTDATA.USER_SUPP_CHALLENGE);

(* Compose 176 bytes of additional authenticated data for use by authenticated decryption *)
AAD := Concatenation of bytes 63:16 and bytes 255:128 of TMP_BIND_STRUCT;

DECRYPT_STRUCT := AES256_GCM_DEC(TMP_BIND_STRUCT.BTENCDATA, WRAPPING_KEY, TMP_BIND_STRUCT.IV, AAD, 176);

(* Fail if MAC mismatch *)
IF TMP_BIND_STRUCT.MAC != DECRYPT_STRUCT.MAC
      THEN
            RFLAGS.ZF := 1;
            RAX := UNWRAP_FAILURE;  (* failure reason 7 *)
            GOTO EXIT;
FI;

Attempt to acquire lock to gain exclusive access to platform key table for TSE;
IF attempt is unsuccessful
      THEN (* PCONFIG failure *)
            RFLAGS.ZF := 1;
            RAX := DEVICE_BUSY;   (* failure reason 5 *)
            GOTO EXIT;
FI;

Update TSE table for RBX[15:0] as follows:
```

```
        Encrypt with the selected key
        Use the encryption algorithm selected by RBX[39:24]
        (* The number of bytes used by the next two lines depends on selected encryption algorithm *)
        DATA_KEY is DECRYPT_STRUCT.DEC_DATA.KEY_FIELD_1
        TWEAK_KEY is DECRYPT_STRUCT.DEC_DATA.KEY_FIELD_2

    Release lock on platform key table for TSE;

ESAC;

RAX := 0;
RFLAGS.ZF := 0;

EXIT:
RFLAGS.CF := 0;
RFLAGS.PF := 0;
RFLAGS.AF := 0;
RFLAGS.OF := 0;
RFLAGS.SF := 0;
```

Protected Mode Exceptions

#GP(0)	If input value in EAX encodes an unsupported leaf function. If a memory operand effective address is outside the relevant segment limit. **MKTME_KEY_PROGRAM leaf function:** If CPUID function 1BH does not enumerate support for the TME-MK target (value 1). If IA32_TME_ACTIVATE MSR is not locked. If hardware encryption and TME-MK capability are not enabled in IA32_TME_ACTIVATE MSR. If the memory operand is not 256B aligned. If any of the reserved bits in the KEYID_CTRL field of the MKTME_KEY_PROGRAM_STRUCT are set or that field indicates an unsupported KeyID, key-programming command, or encryption algorithm. **TSE_KEY_PROGRAM leaf function:** The TSE_KEY_PROGRAM leaf function is not supported in protected mode. **TSE_KEY_PROGRAM_WRAPPED leaf function:** The TSE_KEY_PROGRAM_WRAPPED leaf function is not supported in protected mode.
#PF(fault-code)	If a page fault occurs in accessing memory operands.
#UD	If the LOCK prefix are used. If current privilege level is not 0. If CPUID.07H.00H:EDX.PCONFIG[18] = 0

Real-Address Mode Exceptions

#GP	If input value in EAX encodes an unsupported leaf function. **MKTME_KEY_PROGRAM leaf function:** If CPUID function 1BH does not enumerate support for the TME-MK target (value 1). If IA32_TME_ACTIVATE MSR is not locked. If hardware encryption and TME-MK capability are not enabled in IA32_TME_ACTIVATE MSR. If a memory operand is not 256B aligned. If any of the reserved bits in the KEYID_CTRL field of the MKTME_KEY_PROGRAM_STRUCT are set or that field indicates an unsupported KeyID, key-programming command, or encryption algorithm.

	TSE_KEY_PROGRAM leaf function:
	The TSE_KEY_PROGRAM leaf function is not supported in real-address mode.
	TSE_KEY_PROGRAM_WRAPPED leaf function:
	The TSE_KEY_PROGRAM_WRAPPED leaf function is not supported in real-address mode.
#UD	If the LOCK prefix are used.
	If CPUID.07H.00H:EDX.PCONFIG[18] = 0

Virtual-8086 Mode Exceptions

#UD	PCONFIG instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If input value in EAX encodes an unsupported leaf function.
	If a memory operand is non-canonical form.
	MKTME_KEY_PROGRAM leaf function:
	If CPUID function 1BH does not enumerate support for the TME-MK target (value 1).
	If IA32_TME_ACTIVATE MSR is not locked.
	If hardware encryption and TME-MK capability are not enabled in IA32_TME_ACTIVATE MSR.
	If a memory operand is not 256B aligned.
	If any of the reserved bits in the KEYID_CTRL field of the MKTME_KEY_PROGRAM_STRUCT are set or that field indicates an unsupported KeyID, key-programming command, or encryption algorithm.
	TSE_KEY_PROGRAM leaf function:
	IF CPUID function 1BH does not enumerate support for the TSE target (value 2).
	If RBX is not 256-byte aligned.
	If any of the reserved bits in the KEYID_CTRL field of the TMP_KEY_STRUCT are set or that field indicates an unsupported KeyID, key-programming command, or encryption algorithm.
	TSE_KEY_PROGRAM_WRAPPED leaf function:
	IF CPUID function 1BH does not enumerate support for the TSE target (value 2).
	If RCX is not 256-byte aligned.
	If any of the reserved bits in RBX are set or that register indicates an unsupported KeyID or encryption algorithm.
	If any of the reserved bytes in the TSE_BIND_STRUCT are set (including bytes in BTDATA).
#PF(fault-code)	If a page fault occurs in accessing memory operands.
#UD	If the LOCK prefix are used.
	If the current privilege level is not 0.
	If CPUID.07H.00H:EDX.PCONFIG[18] = 0.

PDEP—Parallel Bits Deposit

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.LZ.F2.0F38.W0 F5 /r PDEP r32a, r32b, r/m32	RVM	V/V	BMI2	Parallel deposit of bits from r32b using mask in r/m32, result is written to r32a.
VEX.LZ.F2.0F38.W1 F5 /r PDEP r64a, r64b, r/m64	RVM	V/N.E.	BMI2	Parallel deposit of bits from r64b using mask in r/m64, result is written to r64a.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

PDEP uses a mask in the second source operand (the third operand) to transfer/scatter contiguous low order bits in the first source operand (the second operand) into the destination (the first operand). PDEP takes the low bits from the first source operand and deposit them in the destination operand at the corresponding bit locations that are set in the second source operand (mask). All other bits (bits not set in mask) in destination are set to zero.

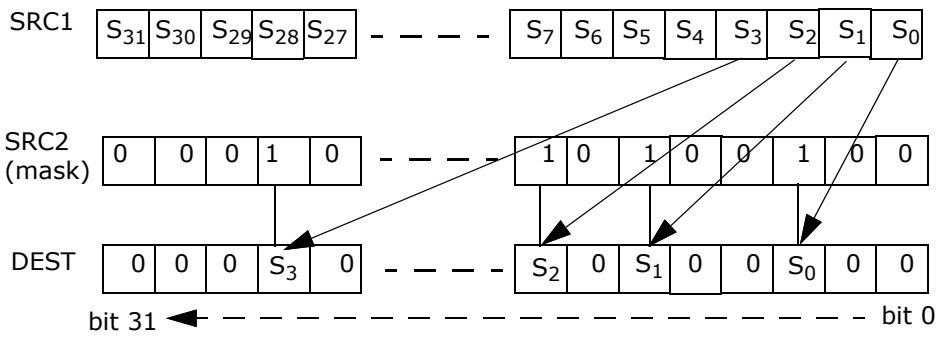


Figure 4-8. PDEP Example

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
TEMP := SRC1;
MASK := SRC2;
DEST := 0 ;
m := 0, k := 0;
DO WHILE m < OperandSize

        IF MASK[ m] = 1 THEN
            DEST[ m] := TEMP[ k];
            k := k+ 1;
        FI
        m := m+ 1;
OD
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

PDEP unsigned __int32 _pdep_u32(unsigned __int32 src, unsigned __int32 mask);
PDEP unsigned __int64 _pdep_u64(unsigned __int64 src, unsigned __int32 mask);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-29, "Type 13 Class Exception Conditions."

PEXT—Parallel Bits Extract

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.LZ.F3.0F38.W0 F5 /r PEXT r32a, r32b, r/m32	RVM	V/V	BMI2	Parallel extract of bits from r32b using mask in r/m32, result is written to r32a.
VEX.LZ.F3.0F38.W1 F5 /r PEXT r64a, r64b, r/m64	RVM	V/N.E.	BMI2	Parallel extract of bits from r64b using mask in r/m64, result is written to r64a.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

PEXT uses a mask in the second source operand (the third operand) to transfer either contiguous or non-contiguous bits in the first source operand (the second operand) to contiguous low order bit positions in the destination (the first operand). For each bit set in the MASK, PEXT extracts the corresponding bits from the first source operand and writes them into contiguous lower bits of destination operand. The remaining upper bits of destination are zeroed.

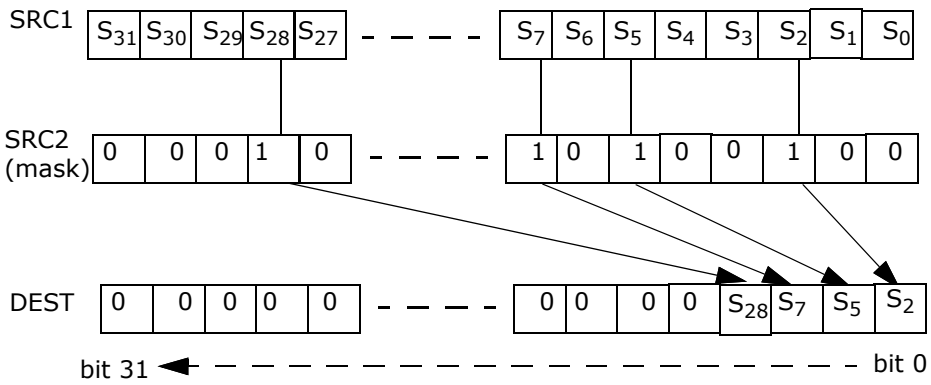


Figure 4-9. PEXT Example

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
TEMP := SRC1;
MASK := SRC2;
DEST := 0 ;
m := 0, k := 0;
DO WHILE m < OperandSize

        IF MASK[ m] = 1 THEN
              DEST[ k] := TEMP[ m];
              k := k+ 1;
        FI
        m := m+ 1;

OD
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

PEXT unsigned __int32 _pext_u32(unsigned __int32 src, unsigned __int32 mask);
PEXT unsigned __int64 _pext_u64(unsigned __int64 src, unsigned __int32 mask);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-29, "Type 13 Class Exception Conditions."

PEXTRB/PEXTRD/PEXTRQ—Extract Byte/Dword/Qword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 14 /r ib PEXTRB reg/m8, xmm2, imm8	A	V/V	SSE4_1	Extract a byte integer value from xmm2 at the source byte offset specified by imm8 into reg or m8. The upper bits of r32 or r64 are zeroed.
66 0F 3A 16 /r ib PEXTRD r/m32, xmm2, imm8	A	V/V	SSE4_1	Extract a dword integer value from xmm2 at the source dword offset specified by imm8 into r/m32.
66 REX.W 0F 3A 16 /r ib PEXTRQ r/m64, xmm2, imm8	A	V/N.E.	SSE4_1	Extract a qword integer value from xmm2 at the source qword offset specified by imm8 into r/m64.
VEX.128.66.0F3A.W0 14 /r ib VPEXTRB reg/m8, xmm2, imm8	A	V[1]/V	AVX	Extract a byte integer value from xmm2 at the source byte offset specified by imm8 into reg or m8. The upper bits of r64/r32 is filled with zeros.
VEX.128.66.0F3A.W0 16 /r ib VPEXTRD r32/m32, xmm2, imm8	A	V/V	AVX	Extract a dword integer value from xmm2 at the source dword offset specified by imm8 into r32/m32.
VEX.128.66.0F3A.W1 16 /r ib VPEXTRQ r64/m64, xmm2, imm8	A	V/I[2]	AVX	Extract a qword integer value from xmm2 at the source dword offset specified by imm8 into r64/m64.
EVEX.128.66.0F3A.WIG 14 /r ib VPEXTRB reg/m8, xmm2, imm8	B	V/V	AVX512BW OR AVX10.1	Extract a byte integer value from xmm2 at the source byte offset specified by imm8 into reg or m8. The upper bits of r64/r32 is filled with zeros.
EVEX.128.66.0F3A.W0 16 /r ib VPEXTRD r32/m32, xmm2, imm8	B	V/V	AVX512DQ OR AVX10.1	Extract a dword integer value from xmm2 at the source dword offset specified by imm8 into r32/m32.
EVEX.128.66.0F3A.W1 16 /r ib VPEXTRQ r64/m64, xmm2, imm8	B	V/N.E.[2]	AVX512DQ OR AVX10.1	Extract a qword integer value from xmm2 at the source dword offset specified by imm8 into r64/m64.

NOTES:

1. In 64-bit mode, VEX.W1 is ignored for VPEXTRB (similar to legacy REX.W=1 prefix in PEXTRB).
2. VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
B	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A

Description

Extract a byte/dword/qword integer value from the source XMM register at a byte/dword/qword offset determined from imm8[3:0]. The destination can be a register or byte/dword/qword memory location. If the destination is a register, the upper bits of the register are zero extended.

In legacy non-VEX encoded version and if the destination operand is a register, the default operand size in 64-bit mode for PEXTRB/PEXTRD is 64 bits, the bits above the least significant byte/dword data are filled with zeros. PEXTRQ is not encodable in non-64-bit modes and requires REX.W in 64-bit mode.

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. In EVEX.128 encoded versions, EVEX.vvvv is reserved and must be 1111b, EVEX.L"L must be 0, otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit mode for VPEXTRB/VPEXTRD is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros.

Operation

```
CASE of
    PEXTRB: SEL := COUNT[3:0];
            TEMP := (Src >> SEL*8) AND FFH;
            IF (DEST = Mem8)
                THEN
                Mem8 := TEMP[7:0];
            ELSE IF (64-Bit Mode and 64-bit register selected)
                THEN
                    R64[7:0] := TEMP[7:0];
                    r64[63:8] := ZERO_FILL; };
            ELSE
                    R32[7:0] := TEMP[7:0];
                    r32[31:8] := ZERO_FILL; };
            FI;
    PEXTRD:SEL := COUNT[1:0];
            TEMP := (Src >> SEL*32) AND FFFF_FFFFH;
            DEST := TEMP;
    PEXTRQ: SEL := COUNT[0];
            TEMP := (Src >> SEL*64);
            DEST := TEMP;
EASC:
```

VPEXTRTD/VPEXTRQ

```
IF (64-Bit Mode and 64-bit dest operand)
THEN
    Src_Offset := imm8[0]
    r64/m64 := (Src >> Src_Offset * 64)
ELSE
    Src_Offset := imm8[1:0]
    r32/m32 := ((Src >> Src_Offset *32) AND 0FFFFFFFFh);
FI
```

VPEXTRB (dest=m8)

```
SRC_Offset := imm8[3:0]
Mem8 := (Src >> Src_Offset*8)
```

VPEXTRB (dest=reg)

```
IF (64-Bit Mode )
THEN
    SRC_Offset := imm8[3:0]
    DEST[7:0] := ((Src >> Src_Offset*8) AND 0FFh)
    DEST[63:8] := ZERO_FILL;
ELSE
    SRC_Offset := imm8[3:0];
    DEST[7:0] := ((Src >> Src_Offset*8) AND 0FFh);
    DEST[31:8] := ZERO_FILL;
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

PEXTRB int _mm_extract_epi8 (__m128i src, const int ndx);
PEXTRD int _mm_extract_epi32 (__m128i src, const int ndx);
PEXTRQ __int64 _mm_extract_epi64 (__m128i src, const int ndx);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-59, “Type E9NF Class Exception Conditions.”
Additionally:

#UD	If VEX.L = 1 or EVEX.L’L > 0.
	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

PEXTRW—Extract Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C5 /r ib[1] PEXTRW reg, mm, imm8	A	V/V	SSE	Extract the word specified by imm8 from mm and move it to reg, bits 15-0. The upper bits of r32 or r64 is zeroed.
66 0F C5 /r ib PEXTRW reg, xmm, imm8	A	V/V	SSE2	Extract the word specified by imm8 from xmm and move it to reg, bits 15-0. The upper bits of r32 or r64 is zeroed.
66 0F 3A 15 /r ib PEXTRW reg/m16, xmm, imm8	B	V/V	SSE4_1	Extract the word specified by imm8 from xmm and copy it to lowest 16 bits of reg or m16. Zero-extend the result in the destination, r32 or r64.
VEX.128.66.0F.W0 C5 /r ib VPEXTRW reg, xmm1, imm8	A	V[2]/V	AVX	Extract the word specified by imm8 from xmm1 and move it to reg, bits 15:0. Zero-extend the result. The upper bits of r64/r32 is filled with zeros.
VEX.128.66.0F3A.W0 15 /r ib VPEXTRW reg/m16, xmm2, imm8	B	V/V	AVX	Extract a word integer value from xmm2 at the source word offset specified by imm8 into reg or m16. The upper bits of r64/r32 is filled with zeros.
EVEX.128.66.0F.WIG C5 /r ib VPEXTRW reg, xmm1, imm8	A	V/V	AVX512BW OR AVX10.1	Extract the word specified by imm8 from xmm1 and move it to reg, bits 15:0. Zero-extend the result. The upper bits of r64/r32 is filled with zeros.
EVEX.128.66.0F3A.WIG 15 /r ib VPEXTRW reg/m16, xmm2, imm8	C	V/V	AVX512BW OR AVX10.1	Extract a word integer value from xmm2 at the source word offset specified by imm8 into reg or m16. The upper bits of r64/r32 is filled with zeros.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.
2. In 64-bit mode, VEX.W1 is ignored for VPEXTRW (similar to legacy REX.W=1 prefix in PEXTRW).

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
C	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A

Description

Copies the word in the source operand (second operand) specified by the count operand (third operand) to the destination operand (first operand). The source operand can be an MMX technology register or an XMM register. The destination operand can be the low word of a general-purpose register or a 16-bit memory address. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location. The content of the destination register above bit 16 is cleared (set to all 0s).

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). If the destination operand is a general-purpose register, the default operand size is 64-bits in 64-bit mode.

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. In EVEX.128 encoded versions, EVEX.vvvv is reserved and must be 1111b, EVEX.L must be 0,

otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit mode for VPEXTRW is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros.

Operation

```
IF (DEST = Mem16)
THEN
    SEL := COUNT[2:0];
    TEMP := (Src >> SEL*16) AND FFFFH;
    Mem16 := TEMP[15:0];
ELSE IF (64-Bit Mode and destination is a general-purpose register)
    THEN
        FOR (PEXTRW instruction with 64-bit source operand)
          {  SEL := COUNT[1:0];
             TEMP := (SRC >> (SEL * 16)) AND FFFFH;
             r64[15:0] := TEMP[15:0];
             r64[63:16] := ZERO_FILL; };
        FOR (PEXTRW instruction with 128-bit source operand)
          {  SEL := COUNT[2:0];
             TEMP := (SRC >> (SEL * 16)) AND FFFFH;
             r64[15:0] := TEMP[15:0];
             r64[63:16] := ZERO_FILL; }
    ELSE
        FOR (PEXTRW instruction with 64-bit source operand)
          {  SEL := COUNT[1:0];
             TEMP := (SRC >> (SEL * 16)) AND FFFFH;
             r32[15:0] := TEMP[15:0];
             r32[31:16] := ZERO_FILL; };
        FOR (PEXTRW instruction with 128-bit source operand)
          {  SEL := COUNT[2:0];
             TEMP := (SRC >> (SEL * 16)) AND FFFFH;
             r32[15:0] := TEMP[15:0];
             r32[31:16] := ZERO_FILL; };
    FI;
FI;
```

VPEXTRW (dest=m16)

```
SRC_Offset := imm8[2:0]
Mem16 := (Src >> Src_Offset*16)
```

VPEXTRW (dest=reg)

```
IF (64-Bit Mode )
THEN
    SRC_Offset := imm8[2:0]
    DEST[15:0] := ((Src >> Src_Offset*16) AND 0FFFFh)
    DEST[63:16] := ZERO_FILL;
ELSE
    SRC_Offset := imm8[2:0]
    DEST[15:0] := ((Src >> Src_Offset*16) AND 0FFFFh)
    DEST[31:16] := ZERO_FILL;
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

PEXTRW int _mm_extract_pi16 (__m64 a, int n)

PEXTRW int _mm_extract_epi16 (__m128i a, int imm)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, "Type 5 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-59, "Type E9NF Class Exception Conditions."
Additionally:

#UD	If VEX.L = 1 or EVEX.L'L > 0.
	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

PHADDSW—Packed Horizontal Add and Saturate

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 03 /r[1] PHADDSW mm1, mm2/m64	RM	V/V	SSSE3	Add 16-bit signed integers horizontally, pack saturated integers to mm1.
66 0F 38 03 /r PHADDSW xmm1, xmm2/m128	RM	V/V	SSSE3	Add 16-bit signed integers horizontally, pack saturated integers to xmm1.
VEX.128.66.0F38.WIG 03 /r VPHADDSW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Add 16-bit signed integers horizontally, pack saturated integers to xmm1.
VEX.256.66.0F38.WIG 03 /r VPHADDSW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Add 16-bit signed integers horizontally, pack saturated integers to ymm1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

(V)PHADDSW adds two adjacent signed 16-bit integers horizontally from the source and destination operands and saturates the signed results; packs the signed, saturated 16-bit results to the destination operand (first operand) When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE version: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Operation

PHADDSW (With 64-bit Operands)

```
mm1[15-0]  = SaturateToSignedWord((mm1[31-16] + mm1[15-0]);
mm1[31-16] = SaturateToSignedWord(mm1[63-48] + mm1[47-32]);
mm1[47-32] = SaturateToSignedWord(mm2/m64[31-16] + mm2/m64[15-0]);
mm1[63-48] = SaturateToSignedWord(mm2/m64[63-48] + mm2/m64[47-32]);
```

PHADDSW (With 128-bit Operands)

```
xmm1[15-0]= SaturateToSignedWord(xmm1[31-16] + xmm1[15-0]);
xmm1[31-16] = SaturateToSignedWord(xmm1[63-48] + xmm1[47-32]);
xmm1[47-32] = SaturateToSignedWord(xmm1[95-80] + xmm1[79-64]);
xmm1[63-48] = SaturateToSignedWord(xmm1[127-112] + xmm1[111-96]);
xmm1[79-64] = SaturateToSignedWord(xmm2/m128[31-16] + xmm2/m128[15-0]);
xmm1[95-80] = SaturateToSignedWord(xmm2/m128[63-48] + xmm2/m128[47-32]);
xmm1[111-96] = SaturateToSignedWord(xmm2/m128[95-80] + xmm2/m128[79-64]);
xmm1[127-112] = SaturateToSignedWord(xmm2/m128[127-112] + xmm2/m128[111-96]);
```

VPHADDSW (VEX.128 Encoded Version)

```
DEST[15:0]= SaturateToSignedWord(SRC1[31:16] + SRC1[15:0])
DEST[31:16] = SaturateToSignedWord(SRC1[63:48] + SRC1[47:32])
DEST[47:32] = SaturateToSignedWord(SRC1[95:80] + SRC1[79:64])
DEST[63:48] = SaturateToSignedWord(SRC1[127:112] + SRC1[111:96])
DEST[79:64] = SaturateToSignedWord(SRC2[31:16] + SRC2[15:0])
DEST[95:80] = SaturateToSignedWord(SRC2[63:48] + SRC2[47:32])
DEST[111:96] = SaturateToSignedWord(SRC2[95:80] + SRC2[79:64])
DEST[127:112] = SaturateToSignedWord(SRC2[127:112] + SRC2[111:96])
DEST[MAXVL-1:128] := 0
```

VPHADDSW (VEX.256 Encoded Version)

```
DEST[15:0]= SaturateToSignedWord(SRC1[31:16] + SRC1[15:0])
DEST[31:16] = SaturateToSignedWord(SRC1[63:48] + SRC1[47:32])
DEST[47:32] = SaturateToSignedWord(SRC1[95:80] + SRC1[79:64])
DEST[63:48] = SaturateToSignedWord(SRC1[127:112] + SRC1[111:96])
DEST[79:64] = SaturateToSignedWord(SRC2[31:16] + SRC2[15:0])
DEST[95:80] = SaturateToSignedWord(SRC2[63:48] + SRC2[47:32])
DEST[111:96] = SaturateToSignedWord(SRC2[95:80] + SRC2[79:64])
DEST[127:112] = SaturateToSignedWord(SRC2[127:112] + SRC2[111:96])
DEST[143:128]= SaturateToSignedWord(SRC1[159:144] + SRC1[143:128])
DEST[159:144] = SaturateToSignedWord(SRC1[191:176] + SRC1[175:160])
DEST[175:160] = SaturateToSignedWord( SRC1[223:208] + SRC1[207:192])
DEST[191:176] = SaturateToSignedWord(SRC1[255:240] + SRC1[239:224])
DEST[207:192] = SaturateToSignedWord(SRC2[127:112] + SRC2[143:128])
DEST[223:208] = SaturateToSignedWord(SRC2[159:144] + SRC2[175:160])
DEST[239:224] = SaturateToSignedWord(SRC2[191-160] + SRC2[159-128])
DEST[255:240] = SaturateToSignedWord(SRC2[255:240] + SRC2[239:224])
```

Intel C/C++ Compiler Intrinsic Equivalent

PHADDSW __m64 _mm_hadds_pi16 (__m64 a, __m64 b)
(V)PHADDSW __m128i _mm_hadds_epi16 (__m128i a, __m128i b)
VPHADDSW __m256i _mm256_hadds_epi16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions," additionally:

#UD	If VEX.L = 1.

PHADDW/PHADDD—Packed Horizontal Add

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 01 /r[1] PHADDW mm1, mm2/m64	RM	V/V	SSSE3	Add 16-bit integers horizontally, pack to mm1.
66 0F 38 01 /r PHADDW xmm1, xmm2/m128	RM	V/V	SSSE3	Add 16-bit integers horizontally, pack to xmm1.
NP 0F 38 02 /r PHADDD mm1, mm2/m64	RM	V/V	SSSE3	Add 32-bit integers horizontally, pack to mm1.
66 0F 38 02 /r PHADDD xmm1, xmm2/m128	RM	V/V	SSSE3	Add 32-bit integers horizontally, pack to xmm1.
VEX.128.66.0F38.WIG 01 /r VPHADDW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Add 16-bit integers horizontally, pack to xmm1.
VEX.128.66.0F38.WIG 02 /r VPHADDD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Add 32-bit integers horizontally, pack to xmm1.
VEX.256.66.0F38.WIG 01 /r VPHADDW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Add 16-bit signed integers horizontally, pack to ymm1.
VEX.256.66.0F38.WIG 02 /r VPHADDD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Add 32-bit signed integers horizontally, pack to ymm1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

(V)PHADDW adds two adjacent 16-bit signed integers horizontally from the source and destination operands and packs the 16-bit signed results to the destination operand (first operand). (V)PHADDD adds two adjacent 32-bit signed integers horizontally from the source and destination operands and packs the 32-bit signed results to the destination operand (first operand). When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Note that these instructions can operate on either unsigned or signed (two's complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values operated on.

Legacy SSE instructions: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: Horizontal addition of two adjacent data elements of the low 16-bytes of the first and second source operands are packed into the low 16-bytes of the destination operand. Horizontal addition of two adjacent data elements of the high 16-bytes of the first and second source operands are packed into the high 16-bytes of the destination operand. The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

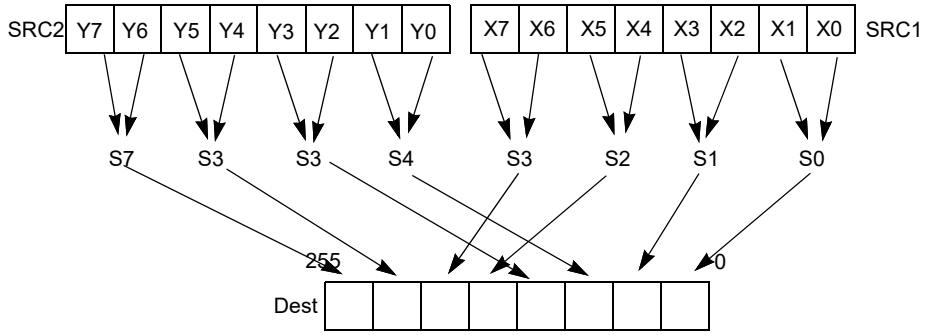


Figure 4-10. 256-bit VPHADDD Instruction Operation

Operation

PHADDW (With 64-bit Operands)

```
mm1[15-0]  = mm1[31-16] + mm1[15-0];
mm1[31-16] = mm1[63-48] + mm1[47-32];
mm1[47-32] = mm2/m64[31-16] + mm2/m64[15-0];
mm1[63-48] = mm2/m64[63-48] + mm2/m64[47-32];
```

PHADDW (With 128-bit Operands)

```
xmm1[15-0] = xmm1[31-16] + xmm1[15-0];
xmm1[31-16] = xmm1[63-48] + xmm1[47-32];
xmm1[47-32] = xmm1[95-80] + xmm1[79-64];
xmm1[63-48] = xmm1[127-112] + xmm1[111-96];
xmm1[79-64] = xmm2/m128[31-16] + xmm2/m128[15-0];
xmm1[95-80] = xmm2/m128[63-48] + xmm2/m128[47-32];
xmm1[111-96] = xmm2/m128[95-80] + xmm2/m128[79-64];
xmm1[127-112] = xmm2/m128[127-112] + xmm2/m128[111-96];
```

VPHADDW (VEX.128 Encoded Version)

```
DEST[15:0] := SRC1[31:16] + SRC1[15:0]
DEST[31:16] := SRC1[63:48] + SRC1[47:32]
DEST[47:32] := SRC1[95:80] + SRC1[79:64]
DEST[63:48] := SRC1[127:112] + SRC1[111:96]
DEST[79:64] := SRC2[31:16] + SRC2[15:0]
DEST[95:80] := SRC2[63:48] + SRC2[47:32]
DEST[111:96] := SRC2[95:80] + SRC2[79:64]
DEST[127:112] := SRC2[127:112] + SRC2[111:96]
DEST[MAXVL-1:128] := 0
```

VPHADDW (VEX.256 Encoded Version)
DEST[15:0] := SRC1[31:16] + SRC1[15:0]
DEST[31:16] := SRC1[63:48] + SRC1[47:32]
DEST[47:32] := SRC1[95:80] + SRC1[79:64]
DEST[63:48] := SRC1[127:112] + SRC1[111:96]
DEST[79:64] := SRC2[31:16] + SRC2[15:0]
DEST[95:80] := SRC2[63:48] + SRC2[47:32]
DEST[111:96] := SRC2[95:80] + SRC2[79:64]
DEST[127:112] := SRC2[127:112] + SRC2[111:96]
DEST[143:128] := SRC1[159:144] + SRC1[143:128]
DEST[159:144] := SRC1[191:176] + SRC1[175:160]
DEST[175:160] := SRC1[223:208] + SRC1[207:192]
DEST[191:176] := SRC1[255:240] + SRC1[239:224]
DEST[207:192] := SRC2[127:112] + SRC2[143:128]
DEST[223:208] := SRC2[159:144] + SRC2[175:160]
DEST[239:224] := SRC2[191:176] + SRC2[207:192]
DEST[255:240] := SRC2[223:208] + SRC2[239:224]

PHADDD (With 64-bit Operands)
mm1[31-0] = mm1[63-32] + mm1[31-0];
mm1[63-32] = mm2/m64[63-32] + mm2/m64[31-0];

PHADDD (With 128-bit Operands)
xmm1[31-0] = xmm1[63-32] + xmm1[31-0];
xmm1[63-32] = xmm1[127-96] + xmm1[95-64];
xmm1[95-64] = xmm2/m128[63-32] + xmm2/m128[31-0];
xmm1[127-96] = xmm2/m128[127-96] + xmm2/m128[95-64];

VPHADDD (VEX.128 Encoded Version)
DEST[31-0] := SRC1[63-32] + SRC1[31-0]
DEST[63-32] := SRC1[127-96] + SRC1[95-64]
DEST[95-64] := SRC2[63-32] + SRC2[31-0]
DEST[127-96] := SRC2[127-96] + SRC2[95-64]
DEST[MAXVL-1:128] := 0

VPHADDD (VEX.256 Encoded Version)
DEST[31-0] := SRC1[63-32] + SRC1[31-0]
DEST[63-32] := SRC1[127-96] + SRC1[95-64]
DEST[95-64] := SRC2[63-32] + SRC2[31-0]
DEST[127-96] := SRC2[127-96] + SRC2[95-64]
DEST[159-128] := SRC1[191-160] + SRC1[159-128]
DEST[191-160] := SRC1[255-224] + SRC1[223-192]
DEST[223-192] := SRC2[191-160] + SRC2[159-128]
DEST[255-224] := SRC2[255-224] + SRC2[223-192]

Intel C/C++ Compiler Intrinsic Equivalents

PHADDW __m64 _mm_hadd_pi16 (__m64 a, __m64 b)
PHADDD __m64 _mm_hadd_pi32 (__m64 a, __m64 b)
(V)PHADDW __m128i _mm_hadd_epi16 (__m128i a, __m128i b)
(V)PHADDD __m128i _mm_hadd_epi32 (__m128i a, __m128i b)
VPHADDW __m256i _mm256_hadd_epi16 (__m256i a, __m256i b)
VPHADDD __m256i _mm256_hadd_epi32 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions,” additionally:

#UD	If VEX.L = 1.

PHMINPOSUW—Packed Horizontal Word Minimum

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 41 /r PHMINPOSUW xmm1, xmm2/m128	RM	V/V	SSE4_1	Find the minimum unsigned word in xmm2/m128 and place its value in the low word of xmm1 and its index in the second-lowest word of xmm1.
VEX.128.66.0F38.WIG 41 /r VPHMINPOSUW xmm1, xmm2/m128	RM	V/V	AVX	Find the minimum unsigned word in xmm2/m128 and place its value in the low word of xmm1 and its index in the second-lowest word of xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Determine the minimum unsigned word value in the source operand (second operand) and place the unsigned word in the low word (bits 0-15) of the destination operand (first operand). The word index of the minimum value is stored in bits 16-18 of the destination operand. The remaining upper bits of the destination are set to zero.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding XMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination XMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

PHMINPOSUW (128-bit Legacy SSE Version)

```
INDEX := 0;
MIN := SRC[15:0]
IF (SRC[31:16] < MIN)
    THEN INDEX := 1;  MIN := SRC[31:16]; FI;
IF (SRC[47:32] < MIN)
    THEN INDEX := 2;  MIN := SRC[47:32]; FI;
* Repeat operation for words 3 through 6
IF (SRC[127:112] < MIN)
    THEN INDEX := 7;  MIN := SRC[127:112]; FI;
DEST[15:0] := MIN;
DEST[18:16] := INDEX;
DEST[127:19] := 00000000000000000000000000000H;
```

VPHMINPOSUW (VEX.128 Encoded Version)

```
INDEX := 0
MIN := SRC[15:0]
IF (SRC[31:16] < MIN) THEN INDEX := 1; MIN := SRC[31:16]
IF (SRC[47:32] < MIN) THEN INDEX := 2; MIN := SRC[47:32]
* Repeat operation for words 3 through 6
IF (SRC[127:112] < MIN) THEN INDEX := 7; MIN := SRC[127:112]
DEST[15:0] := MIN
DEST[18:16] := INDEX
DEST[127:19] := 00000000000000000000000000000H
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PHMINPOSUW __m128i _mm_minpos_epu16(__m128i packed_words);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions," additionally:

#UD	If VEX.L = 1.
	If VEX.vvvv ≠ 1111B.

PHSUBSW—Packed Horizontal Subtract and Saturate

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 07 /r[1] PHSUBSW mm1, mm2/m64	RM	V/V	SSSE3	Subtract 16-bit signed integer horizontally, pack saturated integers to mm1.
66 0F 38 07 /r PHSUBSW xmm1, xmm2/m128	RM	V/V	SSSE3	Subtract 16-bit signed integer horizontally, pack saturated integers to xmm1.
VEX.128.66.0F38.WIG 07 /r VPHSUBSW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Subtract 16-bit signed integer horizontally, pack saturated integers to xmm1.
VEX.256.66.0F38.WIG 07 /r VPHSUBSW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Subtract 16-bit signed integer horizontally, pack saturated integers to ymm1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

(V)PHSUBSW performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands. The signed, saturated 16-bit results are packed to the destination operand (first operand). When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE version: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Operation

PHSUBSW (With 64-bit Operands)

```
mm1[15-0] = SaturateToSignedWord(mm1[15-0] - mm1[31-16]);
mm1[31-16] = SaturateToSignedWord(mm1[47-32] - mm1[63-48]);
mm1[47-32] = SaturateToSignedWord(mm2/m64[15-0] - mm2/m64[31-16]);
mm1[63-48] = SaturateToSignedWord(mm2/m64[47-32] - mm2/m64[63-48]);
```

PHSUBSW (With 128-bit Operands)

```
xmm1[15-0] = SaturateToSignedWord(xmm1[15-0] - xmm1[31-16]);
xmm1[31-16] = SaturateToSignedWord(xmm1[47-32] - xmm1[63-48]);
xmm1[47-32] = SaturateToSignedWord(xmm1[79-64] - xmm1[95-80]);
xmm1[63-48] = SaturateToSignedWord(xmm1[111-96] - xmm1[127-112]);
xmm1[79-64] = SaturateToSignedWord(xmm2/m128[15-0] - xmm2/m128[31-16]);
xmm1[95-80] =SaturateToSignedWord(xmm2/m128[47-32] - xmm2/m128[63-48]);
xmm1[111-96] =SaturateToSignedWord(xmm2/m128[79-64] - xmm2/m128[95-80]);
xmm1[127-112]= SaturateToSignedWord(xmm2/m128[111-96] - xmm2/m128[127-112]);
```

VPHSUBSW (VEX.128 Encoded Version)

```
DEST[15:0]= SaturateToSignedWord(SRC1[15:0] - SRC1[31:16])
DEST[31:16] = SaturateToSignedWord(SRC1[47:32] - SRC1[63:48])
DEST[47:32] = SaturateToSignedWord(SRC1[79:64] - SRC1[95:80])
DEST[63:48] = SaturateToSignedWord(SRC1[111:96] - SRC1[127:112])
DEST[79:64] = SaturateToSignedWord(SRC2[15:0] - SRC2[31:16])
DEST[95:80] = SaturateToSignedWord(SRC2[47:32] - SRC2[63:48])
DEST[111:96] = SaturateToSignedWord(SRC2[79:64] - SRC2[95:80])
DEST[127:112] = SaturateToSignedWord(SRC2[111:96] - SRC2[127:112])
DEST[MAXVL-1:128] := 0
```

VPHSUBSW (VEX.256 Encoded Version)

```
DEST[15:0]= SaturateToSignedWord(SRC1[15:0] - SRC1[31:16])
DEST[31:16] = SaturateToSignedWord(SRC1[47:32] - SRC1[63:48])
DEST[47:32] = SaturateToSignedWord(SRC1[79:64] - SRC1[95:80])
DEST[63:48] = SaturateToSignedWord(SRC1[111:96] - SRC1[127:112])
DEST[79:64] = SaturateToSignedWord(SRC2[15:0] - SRC2[31:16])
DEST[95:80] = SaturateToSignedWord(SRC2[47:32] - SRC2[63:48])
DEST[111:96] = SaturateToSignedWord(SRC2[79:64] - SRC2[95:80])
DEST[127:112] = SaturateToSignedWord(SRC2[111:96] - SRC2[127:112])
DEST[143:128]= SaturateToSignedWord(SRC1[143:128] - SRC1[159:144])
DEST[159:144] = SaturateToSignedWord(SRC1[175:160] - SRC1[191:176])
DEST[175:160] = SaturateToSignedWord(SRC1[207:192] - SRC1[223:208])
DEST[191:176] = SaturateToSignedWord(SRC1[239:224] - SRC1[255:240])
DEST[207:192] = SaturateToSignedWord(SRC2[143:128] - SRC2[159:144])
DEST[223:208] = SaturateToSignedWord(SRC2[175:160] - SRC2[191:176])
DEST[239:224] = SaturateToSignedWord(SRC2[207:192] - SRC2[223:208])
DEST[255:240] = SaturateToSignedWord(SRC2[239:224] - SRC2[255:240])
```

Intel C/C++ Compiler Intrinsic Equivalent

PHSUBSW __m64 _mm_hsubs_pi16 (__m64 a, __m64 b)
(V)PHSUBSW __m128i _mm_hsubs_epi16 (__m128i a, __m128i b)
VPHSUBSW __m256i _mm256_hsubs_epi16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions," additionally:

#UD	If VEX.L = 1.

PHSUBW/PHSUBD—Packed Horizontal Subtract

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 05 /r[1] PHSUBW mm1, mm2/m64	RM	V/V	SSSE3	Subtract 16-bit signed integers horizontally, pack to mm1.
66 0F 38 05 /r PHSUBW xmm1, xmm2/m128	RM	V/V	SSSE3	Subtract 16-bit signed integers horizontally, pack to xmm1.
NP 0F 38 06 /r PHSUBD mm1, mm2/m64	RM	V/V	SSSE3	Subtract 32-bit signed integers horizontally, pack to mm1.
66 0F 38 06 /r PHSUBD xmm1, xmm2/m128	RM	V/V	SSSE3	Subtract 32-bit signed integers horizontally, pack to xmm1.
VEX.128.66.0F38.WIG 05 /r VPHSUBW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Subtract 16-bit signed integers horizontally, pack to xmm1.
VEX.128.66.0F38.WIG 06 /r VPHSUBD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Subtract 32-bit signed integers horizontally, pack to xmm1.
VEX.256.66.0F38.WIG 05 /r VPHSUBW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Subtract 16-bit signed integers horizontally, pack to ymm1.
VEX.256.66.0F38.WIG 06 /r VPHSUBD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Subtract 32-bit signed integers horizontally, pack to ymm1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

(V)PHSUBW performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands, and packs the signed 16-bit results to the destination operand (first operand). (V)PHSUBD performs horizontal subtraction on each adjacent pair of 32-bit signed integers by subtracting the most significant doubleword from the least significant doubleword of each pair, and packs the signed 32-bit result to the destination operand. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE version: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Operation

PHSUBW (With 64-bit Operands)

```
mm1[15-0] = mm1[15-0] - mm1[31-16];
mm1[31-16] = mm1[47-32] - mm1[63-48];
mm1[47-32] = mm2/m64[15-0] - mm2/m64[31-16];
mm1[63-48] = mm2/m64[47-32] - mm2/m64[63-48];
```

PHSUBW (With 128-bit Operands)

```
xmm1[15-0] = xmm1[15-0] - xmm1[31-16];
xmm1[31-16] = xmm1[47-32] - xmm1[63-48];
xmm1[47-32] = xmm1[79-64] - xmm1[95-80];
xmm1[63-48] = xmm1[111-96] - xmm1[127-112];
xmm1[79-64] = xmm2/m128[15-0] - xmm2/m128[31-16];
xmm1[95-80] = xmm2/m128[47-32] - xmm2/m128[63-48];
xmm1[111-96] = xmm2/m128[79-64] - xmm2/m128[95-80];
xmm1[127-112] = xmm2/m128[111-96] - xmm2/m128[127-112];
```

VPHSUBW (VEX.128 Encoded Version)

```
DEST[15:0] := SRC1[15:0] - SRC1[31:16]
DEST[31:16] := SRC1[47:32] - SRC1[63:48]
DEST[47:32] := SRC1[79:64] - SRC1[95:80]
DEST[63:48] := SRC1[111:96] - SRC1[127:112]
DEST[79:64] := SRC2[15:0] - SRC2[31:16]
DEST[95:80] := SRC2[47:32] - SRC2[63:48]
DEST[111:96] := SRC2[79:64] - SRC2[95:80]
DEST[127:112] := SRC2[111:96] - SRC2[127:112]
DEST[MAXVL-1:128] := 0
```

VPHSUBW (VEX.256 Encoded Version)

```
DEST[15:0] := SRC1[15:0] - SRC1[31:16]
DEST[31:16] := SRC1[47:32] - SRC1[63:48]
DEST[47:32] := SRC1[79:64] - SRC1[95:80]
DEST[63:48] := SRC1[111:96] - SRC1[127:112]
DEST[79:64] := SRC2[15:0] - SRC2[31:16]
DEST[95:80] := SRC2[47:32] - SRC2[63:48]
DEST[111:96] := SRC2[79:64] - SRC2[95:80]
DEST[127:112] := SRC2[111:96] - SRC2[127:112]
DEST[143:128] := SRC1[143:128] - SRC1[159:144]
DEST[159:144] := SRC1[175:160] - SRC1[191:176]
DEST[175:160] := SRC1[207:192] - SRC1[223:208]
DEST[191:176] := SRC1[239:224] - SRC1[255:240]
DEST[207:192] := SRC2[143:128] - SRC2[159:144]
DEST[223:208] := SRC2[175:160] - SRC2[191:176]
DEST[239:224] := SRC2[207:192] - SRC2[223:208]
DEST[255:240] := SRC2[239:224] - SRC2[255:240]
```

PHSUBD (With 64-bit Operands)

```
mm1[31-0] = mm1[31-0] - mm1[63-32];
mm1[63-32] = mm2/m64[31-0] - mm2/m64[63-32];
```

PHSUBD (With 128-bit Operands)

```
xmm1[31-0] = xmm1[31-0] - xmm1[63-32];
xmm1[63-32] = xmm1[95-64] - xmm1[127-96];
xmm1[95-64] = xmm2/m128[31-0] - xmm2/m128[63-32];
xmm1[127-96] = xmm2/m128[95-64] - xmm2/m128[127-96];
```

VPHSUBD (VEX.128 Encoded Version)

```
DEST[31-0] := SRC1[31-0] - SRC1[63-32]
DEST[63-32] := SRC1[95-64] - SRC1[127-96]
DEST[95-64] := SRC2[31-0] - SRC2[63-32]
DEST[127-96] := SRC2[95-64] - SRC2[127-96]
DEST[MAXVL-1:128] := 0
```

VPHSUBD (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[31:0] - SRC1[63:32]
DEST[63:32] := SRC1[95:64] - SRC1[127:96]
DEST[95:64] := SRC2[31:0] - SRC2[63:32]
DEST[127:96] := SRC2[95:64] - SRC2[127:96]
DEST[159:128] := SRC1[159:128] - SRC1[191:160]
DEST[191:160] := SRC1[223:192] - SRC1[255:224]
DEST[223:192] := SRC2[159:128] - SRC2[191:160]
DEST[255:224] := SRC2[223:192] - SRC2[255:224]
```

Intel C/C++ Compiler Intrinsic Equivalents

PHSUBW __m64 _mm_hsub_pi16 (__m64 a, __m64 b)
PHSUBD __m64 _mm_hsub_pi32 (__m64 a, __m64 b)
(V)PHSUBW __m128i _mm_hsub_epi16 (__m128i a, __m128i b)
(V)PHSUBD __m128i _mm_hsub_epi32 (__m128i a, __m128i b)
VPHSUBW __m256i _mm256_hsub_epi16 (__m256i a, __m256i b)
VPHSUBD __m256i _mm256_hsub_epi32 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions," additionally:

#UD	If VEX.L = 1.

PINSRB/PINSRD/PINSRQ—Insert Byte/Dword/Qword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 20 /r ib PINSRB xmm1, r32/m8, imm8	A	V/V	SSE4_1	Insert a byte integer value from r32/m8 into xmm1 at the destination element in xmm1 specified by imm8.
66 0F 3A 22 /r ib PINSRD xmm1, r/m32, imm8	A	V/V	SSE4_1	Insert a dword integer value from r/m32 into the xmm1 at the destination element specified by imm8.
66 REX.W 0F 3A 22 /r ib PINSRQ xmm1, r/m64, imm8	A	V/N. E.	SSE4_1	Insert a qword integer value from r/m64 into the xmm1 at the destination element specified by imm8.
VEX.128.66.0F3A.W0 20 /r ib VPINSRB xmm1, xmm2, r32/m8, imm8	B	V[1]/V	AVX	Merge a byte integer value from r32/m8 and rest from xmm2 into xmm1 at the byte offset in imm8.
VEX.128.66.0F3A.W0 22 /r ib VPINSRD xmm1, xmm2, r/m32, imm8	B	V/V	AVX	Insert a dword integer value from r32/m32 and rest from xmm2 into xmm1 at the dword offset in imm8.
VEX.128.66.0F3A.W1 22 /r ib VPINSRQ xmm1, xmm2, r/m64, imm8	B	V/I[2]	AVX	Insert a qword integer value from r64/m64 and rest from xmm2 into xmm1 at the qword offset in imm8.
EVEX.128.66.0F3A.WIG 20 /r ib VPINSRB xmm1, xmm2, r32/m8, imm8	C	V/V	AVX512BW OR AVX10.1	Merge a byte integer value from r32/m8 and rest from xmm2 into xmm1 at the byte offset in imm8.
EVEX.128.66.0F3A.W0 22 /r ib VPINSRD xmm1, xmm2, r32/m32, imm8	C	V/V	AVX512DQ OR AVX10.1	Insert a dword integer value from r32/m32 and rest from xmm2 into xmm1 at the dword offset in imm8.
EVEX.128.66.0F3A.W1 22 /r ib VPINSRQ xmm1, xmm2, r64/m64, imm8	C	V/N.E.[2]	AVX512DQ OR AVX10.1	Insert a qword integer value from r64/m64 and rest from xmm2 into xmm1 at the qword offset in imm8.

NOTES:

1. In 64-bit mode, VEX.W1 is ignored for VPINSRB (similar to legacy REX.W=1 prefix with PINSRB).
2. VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Copies a byte/dword/qword from the source operand (second operand) and inserts it in the destination operand (first operand) at the location specified with the count operand (third operand). (The other elements in the destination register are left untouched.) The source operand can be a general-purpose register or a memory location. (When the source operand is a general-purpose register, PINSRB copies the low byte of the register.) The destination operand is an XMM register. The count operand is an 8-bit immediate. When specifying a qword[dword, byte] location in an XMM register, the [2, 4] least-significant bit(s) of the count operand specify the location.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). Use of REX.W permits the use of 64 bit general purpose registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. VEX.L must be 0, otherwise the instruction will #UD. Attempt to execute VPINSRQ in non-64-bit mode will cause #UD.

EVEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. EVEX.L'L must be 0, otherwise the instruction will #UD.

Operation

```
CASE OF
    PINSRB:  SEL := COUNT[3:0];
             MASK := (0FFH << (SEL * 8));
             TEMP := (((SRC[7:0] << (SEL *8)) AND MASK);
    PINSRD:  SEL := COUNT[1:0];
             MASK := (0FFFFFFFFH << (SEL * 32));
             TEMP := (((SRC << (SEL *32)) AND MASK)   ;
    PINSRQ:  SEL := COUNT[0]
             MASK := (0FFFFFFFFFFFFFFFFH << (SEL * 64));
             TEMP := (((SRC << (SEL *64)) AND MASK)   ;
ESAC;
        DEST := ((DEST AND NOT MASK) OR TEMP);
```

VPINSRB (VEX/EVEX Encoded Version)

```
SEL := imm8[3:0]
DEST[127:0] := write_b_element(SEL, SRC2, SRC1)
DEST[MAXVL-1:128] := 0
```

VPINSRD (VEX/EVEX Encoded Version)

```
SEL := imm8[1:0]
DEST[127:0] := write_d_element(SEL, SRC2, SRC1)
DEST[MAXVL-1:128] := 0
```

VPINSRQ (VEX/EVEX Encoded Version)

```
SEL := imm8[0]
DEST[127:0] := write_q_element(SEL, SRC2, SRC1)
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PINSRB __m128i _mm_insert_epi8 (__m128i s1, int s2, const int ndx);
PINSRD __m128i _mm_insert_epi32 (__m128i s2, int s, const int ndx);
PINSRQ __m128i _mm_insert_epi64(__m128i s2, __int64 s, const int ndx);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-59, “Type E9NF Class Exception Conditions.”
Additionally:

#UD	If VEX.L = 1 or EVEX.L’L > 0.

PINSRW—Insert Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C4 /*r* ib[1] PINSRW mm, r32/m16, imm8	A	V/V	SSE	Insert the low word from r32 or from m16 into mm at the word position specified by imm8.
66 0F C4 /*r* ib PINSRW xmm, r32/m16, imm8	A	V/V	SSE2	Move the low word of r32 or from m16 into xmm at the word position specified by imm8.
VEX.128.66.0F.W0 C4 /r ib VPINSRW xmm1, xmm2, r32/m16, imm8	B	V[2]/V	AVX	Insert the word from r32/m16 at the offset indicated by imm8 into the value from xmm2 and store result in xmm1.
EVEX.128.66.0F.WIG C4 /r ib VPINSRW xmm1, xmm2, r32/m16, imm8	C	V/V	AVX512BW OR AVX10.1	Insert the word from r32/m16 at the offset indicated by imm8 into the value from xmm2 and store result in xmm1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.
2. In 64-bit mode, VEX.W1 is ignored for VPINSRW (similar to legacy REX.W=1 prefix in PINSRW).

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Three operand MMX and SSE instructions:

Copies a word from the source operand and inserts it in the destination operand at the location specified with the count operand. (The other words in the destination register are left untouched.) The source operand can be a general-purpose register or a 16-bit memory location. (When the source operand is a general-purpose register, the low word of the register is copied.) The destination operand can be an MMX technology register or an XMM register. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location.

Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

Four operand AVX and AVX-512 instructions:

Combines a word from the first source operand with the second source operand, and inserts it in the destination operand at the location specified with the count operand. The second source operand can be a general-purpose register or a 16-bit memory location. (When the source operand is a general-purpose register, the low word of the register is copied.) The first source and destination operands are XMM registers. The count operand is an 8-bit immediate. When specifying a word location, the 3 least-significant bits specify the location.

Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L/EVEX.L'L must be 0, otherwise the instruction will #UD.

Operation

PINSRW dest, src, imm8 (MMX)

```
SEL := imm8[1:0]
DEST.word[SEL] := src.word[0]
```

PINSRW dest, src, imm8 (SSE)

```
SEL := imm8[2:0]
DEST.word[SEL] := src.word[0]
```

VPINSRW dest, src1, src2, imm8 (AVX/AVX512)

```
SEL := imm8[2:0]
DEST := src1
DEST.word[SEL] := src2.word[0]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PINSRW __m64 _mm_insert_pi16 (__m64 a, int d, int n)
PINSRW __m128i _mm_insert_epi16 (__m128i a, int b, int imm)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-59, “Type E9NF Class Exception Conditions.”
Additionally:

#UD	If VEX.L = 1 or EVEX.L’L > 0.

PMADDUBSW—Multiply and Add Packed Signed and Unsigned Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 04 /r[1] PMADDUBSW mm1, mm2/m64	A	V/V	SSSE3	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to mm1.
66 0F 38 04 /r PMADDUBSW xmm1, xmm2/m128	A	V/V	SSSE3	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to xmm1.
VEX.128.66.0F38.WIG 04 /r VPMADDUBSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to xmm1.
VEX.256.66.0F38.WIG 04 /r VPMADDUBSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to ymm1.
EVEX.128.66.0F38.WIG 04 /r VPMADDUBSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to xmm1 under writemask k1.
EVEX.256.66.0F38.WIG 04 /r VPMADDUBSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to ymm1 under writemask k1.
EVEX.512.66.0F38.WIG 04 /r VPMADDUBSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

(V)PMADDUBSW multiplies vertically each unsigned byte of the destination operand (first operand) with the corresponding signed byte of the source operand (second operand), producing intermediate signed 16-bit integers. Each adjacent pair of signed words is added and the saturated result is packed to the destination operand. For example, the lowest-order bytes (bits 7-0) in the source and destination operands are multiplied and the intermediate signed word result is added with the corresponding intermediate result from the 2nd lowest-order bytes (bits 15-8) of the operands; the sign-saturated result is stored in the lowest word of the destination register (15-0). The same operation is performed on the other pairs of adjacent bytes. Both operands can be MMX register or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The second source operand can be an ZMM register or a 512-bit memory location. The first source and destination operands are ZMM registers.

Operation

PMADDUBSW (With 64-bit Operands)

```
DEST[15-0] = SaturateToSignedWord(SRC[15-8]*DEST[15-8]+SRC[7-0]*DEST[7-0]);
DEST[31-16] = SaturateToSignedWord(SRC[31-24]*DEST[31-24]+SRC[23-16]*DEST[23-16]);
DEST[47-32] = SaturateToSignedWord(SRC[47-40]*DEST[47-40]+SRC[39-32]*DEST[39-32]);
DEST[63-48] = SaturateToSignedWord(SRC[63-56]*DEST[63-56]+SRC[55-48]*DEST[55-48]);
```

PMADDUBSW (With 128-bit Operands)

```
DEST[15-0] = SaturateToSignedWord(SRC[15-8]* DEST[15-8]+SRC[7-0]*DEST[7-0]);
// Repeat operation for 2nd through 7th word
SRC1/DEST[127-112] = SaturateToSignedWord(SRC[127-120]*DEST[127-120]+ SRC[119-112]* DEST[119-112]);
```

VPMADDUBSW (VEX.128 Encoded Version)

```
DEST[15:0] := SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])
// Repeat operation for 2nd through 7th word
DEST[127:112] := SaturateToSignedWord(SRC2[127:120]*SRC1[127:120]+ SRC2[119:112]* SRC1[119:112])
DEST[MAXVL-1:128] := 0
```

VPMADDUBSW (VEX.256 Encoded Version)

```
DEST[15:0] := SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])
// Repeat operation for 2nd through 15th word
DEST[255:240] := SaturateToSignedWord(SRC2[255:248]*SRC1[255:248]+ SRC2[247:240]* SRC1[247:240])
DEST[MAXVL-1:256] := 0
```

VPMADDUBSW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SaturateToSignedWord(SRC2[i+15:i+8]* SRC1[i+15:i+8] + SRC2[i+7:i]*SRC1[i+7:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPMADDUBSW __m512i _mm512_maddubs_epi16( __m512i a, __m512i b);
VPMADDUBSW __m512i _mm512_mask_maddubs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
```

VPMADDUBSW __m512i _mm512_maskz_maddubs_epi16(__mmask32 k, __m512i a, __m512i b);
VPMADDUBSW __m256i _mm256_mask_maddubs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMADDUBSW __m256i _mm256_maskz_maddubs_epi16(__mmask16 k, __m256i a, __m256i b);
VPMADDUBSW __m128i _mm_mask_maddubs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMADDUBSW __m128i _mm_maskz_maddubs_epi16(__mmask8 k, __m128i a, __m128i b);
PMADDUBSW __m64 _mm_maddubs_pi16 (__m64 a, __m64 b)
(V)PMADDUBSW __m128i _mm_maddubs_epi16 (__m128i a, __m128i b)
VPMADDUBSW __m256i _mm256_maddubs_epi16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."

PMADDWD—Multiply and Add Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F5 /r[1] PMADDWD mm, mm/m64	A	V/V	MMX	Multiply the packed words in mm by the packed words in mm/m64, add adjacent doubleword results, and store in mm.
66 0F F5 /r PMADDWD xmm1, xmm2/m128	A	V/V	SSE2	Multiply the packed word integers in xmm1 by the packed word integers in xmm2/m128, add adjacent doubleword results, and store in xmm1.
VEX.128.66.0F.WIG F5 /r VPMADDWD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed word integers in xmm2 by the packed word integers in xmm3/m128, add adjacent doubleword results, and store in xmm1.
VEX.256.66.0F.WIG F5 /r VPMADDWD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply the packed word integers in ymm2 by the packed word integers in ymm3/m256, add adjacent doubleword results, and store in ymm1.
EVEX.128.66.0F.WIG F5 /r VPMADDWD xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Multiply the packed word integers in xmm2 by the packed word integers in xmm3/m128, add adjacent doubleword results, and store in xmm1 under writemask k1.
EVEX.256.66.0F.WIG F5 /r VPMADDWD ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Multiply the packed word integers in ymm2 by the packed word integers in ymm3/m256, add adjacent doubleword results, and store in ymm1 under writemask k1.
EVEX.512.66.0F.WIG F5 /r VPMADDWD zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Multiply the packed word integers in zmm2 by the packed word integers in zmm3/m512, add adjacent doubleword results, and store in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies the individual signed words of the destination operand (first operand) by the corresponding signed words of the source operand (second operand), producing temporary signed, doubleword results. The adjacent doubleword results are then summed and stored in the destination operand. For example, the corresponding low-order words (15-0) and (31-16) in the source and destination operands are multiplied by one another and the doubleword results are added together and stored in the low doubleword of the destination register (31-0). The same operation is performed on the other pairs of adjacent words. (Figure 4-11 shows this operation when using 64-bit operands).

The (V)PMADDWD instruction wraps around only in one situation: when the 2 pairs of words being operated on in a group are all 8000H. In this case, the result wraps around to 80000000H.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The first source and destination operands are MMX registers. The second source operand is an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX.512 encoded version: The second source operand can be an ZMM register or a 512-bit memory location. The first source and destination operands are ZMM registers.

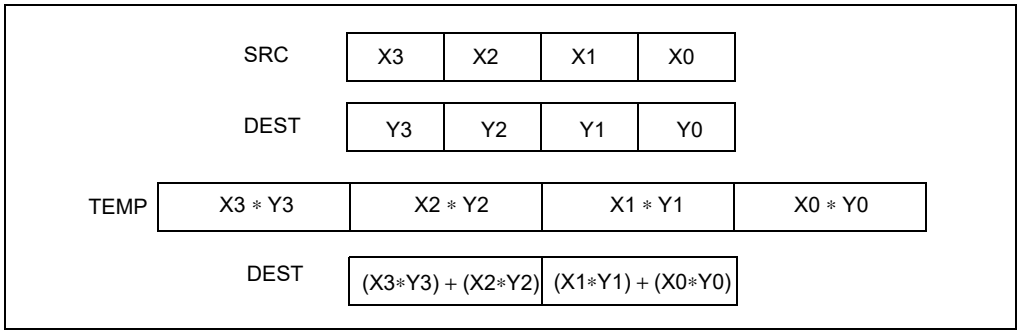


Figure 4-11. PMADDWD Execution Model Using 64-bit Operands

Operation

PMADDWD (With 64-bit Operands)

```
DEST[31:0] := (DEST[15:0] * SRC[15:0]) + (DEST[31:16] * SRC[31:16]);
DEST[63:32] := (DEST[47:32] * SRC[47:32]) + (DEST[63:48] * SRC[63:48]);
```

PMADDWD (With 128-bit Operands)

```
DEST[31:0] := (DEST[15:0] * SRC[15:0]) + (DEST[31:16] * SRC[31:16]);
DEST[63:32] := (DEST[47:32] * SRC[47:32]) + (DEST[63:48] * SRC[63:48]);
DEST[95:64] := (DEST[79:64] * SRC[79:64]) + (DEST[95:80] * SRC[95:80]);
DEST[127:96] := (DEST[111:96] * SRC[111:96]) + (DEST[127:112] * SRC[127:112]);
```

VPMADDWD (VEX.128 Encoded Version)

```
DEST[31:0] := (SRC1[15:0] * SRC2[15:0]) + (SRC1[31:16] * SRC2[31:16])
DEST[63:32] := (SRC1[47:32] * SRC2[47:32]) + (SRC1[63:48] * SRC2[63:48])
DEST[95:64] := (SRC1[79:64] * SRC2[79:64]) + (SRC1[95:80] * SRC2[95:80])
DEST[127:96] := (SRC1[111:96] * SRC2[111:96]) + (SRC1[127:112] * SRC2[127:112])
DEST[MAXVL-1:128] := 0
```

VPMADDWD (VEX.256 Encoded Version)

```
DEST[31:0] := (SRC1[15:0] * SRC2[15:0]) + (SRC1[31:16] * SRC2[31:16])
DEST[63:32] := (SRC1[47:32] * SRC2[47:32]) + (SRC1[63:48] * SRC2[63:48])
DEST[95:64] := (SRC1[79:64] * SRC2[79:64]) + (SRC1[95:80] * SRC2[95:80])
DEST[127:96] := (SRC1[111:96] * SRC2[111:96]) + (SRC1[127:112] * SRC2[127:112])
DEST[159:128] := (SRC1[143:128] * SRC2[143:128]) + (SRC1[159:144] * SRC2[159:144])
DEST[191:160] := (SRC1[175:160] * SRC2[175:160]) + (SRC1[191:176] * SRC2[191:176])
DEST[223:192] := (SRC1[207:192] * SRC2[207:192]) + (SRC1[223:208] * SRC2[223:208])
DEST[255:224] := (SRC1[239:224] * SRC2[239:224]) + (SRC1[255:240] * SRC2[255:240])
DEST[MAXVL-1:256] := 0
```

VPMADDWD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := (SRC2[i+31:i+16]* SRC1[i+31:i+16]) + (SRC2[i+15:i]*SRC1[i+15:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMADDWD __m512i _mm512_madd_epi16(__m512i a, __m512i b);
VPMADDWD __m512i _mm512_mask_madd_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMADDWD __m512i _mm512_maskz_madd_epi16(__mmask32 k, __m512i a, __m512i b);
VPMADDWD __m256i _mm256_mask_madd_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMADDWD __m256i _mm256_maskz_madd_epi16(__mmask16 k, __m256i a, __m256i b);
VPMADDWD __m128i _mm_mask_madd_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMADDWD __m128i _mm_maskz_madd_epi16(__mmask8 k, __m128i a, __m128i b);
PMADDWD __m64 _mm_madd_pi16(__m64 m1, __m64 m2)
(V)PMADDWD __m128i _mm_madd_epi16 (__m128i a, __m128i b)
VPMADDWD __m256i _mm256_madd_epi16 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."

PMAXSB/PMAXSW/PMAXSD/PMAXSQ—Maximum of Packed Signed Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EE /r[1] PMAXSW mm1, mm2/m64	A	V/V	SSE	Compare signed word integers in mm2/m64 and mm1 and return maximum values.
66 0F 38 3C /r PMAXSB xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
66 0F EE /r PMAXSW xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1.
66 0F 38 3D /r PMAXSD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed dword integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
VEX.128.66.0F38.WIG 3C /r VPMAXSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.128.66.0F.WIG EE /r VPMAXSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed word integers in xmm3/m128 and xmm2 and store packed maximum values in xmm1.
VEX.128.66.0F38.WIG 3D /r VPMAXSD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.256.66.0F38.WIG 3C /r VPMAXSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
VEX.256.66.0F.WIG EE /r VPMAXSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed word integers in ymm3/m256 and ymm2 and store packed maximum values in ymm1.
VEX.256.66.0F38.WIG 3D /r VPMAXSD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed dword integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
EVEX.128.66.0F38.WIG 3C /r VPMAXSB xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.256.66.0F38.WIG 3C /r VPMAXSB ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.512.66.0F38.WIG 3C /r VPMAXSB zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Compare packed signed byte integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.
EVEX.128.66.0F.WIG EE /r VPMAXSW xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed word integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.256.66.0F.WIG EE /r VPMAXSW ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed word integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.512.66.0F.WIG EE /r VPMAXSW zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Compare packed signed word integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 3D /r VPMAXSD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed signed dword integers in xmm2 and xmm3/m128/m32bcst and store packed maximum values in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 3D /r VPMAXSD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed signed dword integers in ymm2 and ymm3/m256/m32bcst and store packed maximum values in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 3D /r VPMAXSD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F OR AVX10.1	Compare packed signed dword integers in zmm2 and zmm3/m512/m32bcst and store packed maximum values in zmm1 using writemask k1.
EVEX.128.66.0F38.W1 3D /r VPMAXSQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed signed qword integers in xmm2 and xmm3/m128/m64bcst and store packed maximum values in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 3D /r VPMAXSQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed signed qword integers in ymm2 and ymm3/m256/m64bcst and store packed maximum values in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 3D /r VPMAXSQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	D	V/V	AVX512F OR AVX10.1	Compare packed signed qword integers in zmm2 and zmm3/m512/m64bcst and store packed maximum values in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed signed byte, word, dword or qword integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

Legacy SSE version PMAXSW: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded VPMAXSD/Q: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

EVEX encoded VPMAXSB/W: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMAXSW (64-bit Operands)

```
    IF DEST[15:0] > SRC[15:0]) THEN
        DEST[15:0] := DEST[15:0];
    ELSE
        DEST[15:0] := SRC[15:0]; FI;
    (* Repeat operation for 2nd and 3rd words in source and destination operands *)
    IF DEST[63:48] > SRC[63:48]) THEN
        DEST[63:48] := DEST[63:48];
    ELSE
        DEST[63:48] := SRC[63:48]; FI;
```

PMAXSB (128-bit Legacy SSE Version)

```
    IF DEST[7:0] > SRC[7:0] THEN
        DEST[7:0] := DEST[7:0];
    ELSE
        DEST[7:0] := SRC[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF DEST[127:120] >SRC[127:120] THEN
        DEST[127:120] := DEST[127:120];
    ELSE
        DEST[127:120] := SRC[127:120]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXSB (VEX.128 Encoded Version)

```
    IF SRC1[7:0] > SRC2[7:0] THEN
        DEST[7:0] := SRC1[7:0];
    ELSE
        DEST[7:0] := SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF SRC1[127:120] >SRC2[127:120] THEN
        DEST[127:120] := SRC1[127:120];
    ELSE
        DEST[127:120] := SRC2[127:120]; FI;
DEST[MAXVL-1:128] := 0
```

VPMAXSB (VEX.256 Encoded Version)

```
    IF SRC1[7:0] > SRC2[7:0] THEN
        DEST[7:0] := SRC1[7:0];
    ELSE
        DEST[7:0] := SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    IF SRC1[255:248] >SRC2[255:248] THEN
        DEST[255:248] := SRC1[255:248];
    ELSE
        DEST[255:248] := SRC2[255:248]; FI;
DEST[MAXVL-1:256] := 0
```

VPMAXSB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] > SRC2[i+7:i]
            THEN DEST[i+7:i] := SRC1[i+7:i];
            ELSE DEST[i+7:i] := SRC2[i+7:i];
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

PMAXSW (128-bit Legacy SSE Version)

```
    IF DEST[15:0] >SRC[15:0] THEN
        DEST[15:0] := DEST[15:0];
    ELSE
        DEST[15:0] := SRC[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:112] >SRC[127:112] THEN
        DEST[127:112] := DEST[127:112];
    ELSE
        DEST[127:112] := SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXSW (VEX.128 Encoded Version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] := SRC1[15:0];
    ELSE
        DEST[15:0] := SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] >SRC2[127:112] THEN
        DEST[127:112] := SRC1[127:112];
    ELSE
        DEST[127:112] := SRC2[127:112]; FI;
DEST[MAXVL-1:128] := 0
```

VPMAXSW (VEX.256 Encoded Version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] := SRC1[15:0];
    ELSE
        DEST[15:0] := SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] >SRC2[255:240] THEN
        DEST[255:240] := SRC1[255:240];
    ELSE
        DEST[255:240] := SRC2[255:240]; FI;
DEST[MAXVL-1:256] := 0
```

VPMAXSW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+15:i] > SRC2[i+15:i]
            THEN DEST[i+15:i] := SRC1[i+15:i];
            ELSE DEST[i+15:i] := SRC2[i+15:i];
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

PMAXSD (128-bit Legacy SSE Version)

```
    IF DEST[31:0] >SRC[31:0] THEN
        DEST[31:0] := DEST[31:0];
    ELSE
        DEST[31:0] := SRC[31:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:96] >SRC[127:96] THEN
        DEST[127:96] := DEST[127:96];
    ELSE
        DEST[127:96] := SRC[127:96]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXSD (VEX.128 Encoded Version)

```
    IF SRC1[31:0] > SRC2[31:0] THEN
        DEST[31:0] := SRC1[31:0];
    ELSE
        DEST[31:0] := SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
    IF SRC1[127:96] > SRC2[127:96] THEN
        DEST[127:96] := SRC1[127:96];
    ELSE
        DEST[127:96] := SRC2[127:96]; FI;
DEST[MAXVL-1:128] := 0
```

VPMAXSD (VEX.256 Encoded Version)

```
    IF SRC1[31:0] > SRC2[31:0] THEN
        DEST[31:0] := SRC1[31:0];
    ELSE
        DEST[31:0] := SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 7th dwords in source and destination operands *)
    IF SRC1[255:224] > SRC2[255:224] THEN
        DEST[255:224] := SRC1[255:224];
    ELSE
        DEST[255:224] := SRC2[255:224]; FI;
DEST[MAXVL-1:256] := 0
```

VPMAXSD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+31:i] > SRC2[31:0]
                    THEN DEST[i+31:i] := SRC1[i+31:i];
                    ELSE DEST[i+31:i] := SRC2[31:0];
                FI;
            ELSE
                IF SRC1[i+31:i] > SRC2[i+31:i]
                    THEN DEST[i+31:i] := SRC1[i+31:i];
                    ELSE DEST[i+31:i] := SRC2[i+31:i];
            FI;
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] := 0             ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMAXSQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+63:i] > SRC2[63:0]
                    THEN DEST[i+63:i] := SRC1[i+63:i];
                    ELSE DEST[i+63:i] := SRC2[63:0];
                FI;
            ELSE
                IF SRC1[i+63:i] > SRC2[i+63:i]
                    THEN DEST[i+63:i] := SRC1[i+63:i];
                    ELSE DEST[i+63:i] := SRC2[i+63:i];
            FI;
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    THEN DEST[i+63:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMAXSB __m512i _mm512_max_epi8(__m512i a, __m512i b);
VPMAXSB __m512i _mm512_mask_max_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMAXSB __m512i _mm512_maskz_max_epi8(__mmask64 k, __m512i a, __m512i b);
VPMAXSW __m512i _mm512_max_epi16(__m512i a, __m512i b);
VPMAXSW __m512i _mm512_mask_max_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMAXSW __m512i _mm512_maskz_max_epi16(__mmask32 k, __m512i a, __m512i b);
VPMAXSB __m256i _mm256_mask_max_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMAXSB __m256i _mm256_maskz_max_epi8(__mmask32 k, __m256i a, __m256i b);
VPMAXSW __m256i _mm256_mask_max_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXSW __m256i _mm256_maskz_max_epi16(__mmask16 k, __m256i a, __m256i b);
VPMAXSB __m128i _mm_mask_max_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMAXSB __m128i _mm_maskz_max_epi8(__mmask16 k, __m128i a, __m128i b);
VPMAXSW __m128i _mm_mask_max_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXSW __m128i _mm_maskz_max_epi16(__mmask8 k, __m128i a, __m128i b);
VPMAXSD __m256i _mm256_mask_max_epi32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXSD __m256i _mm256_maskz_max_epi32(__mmask16 k, __m256i a, __m256i b);
VPMAXSQ __m256i _mm256_mask_max_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMAXSQ __m256i _mm256_maskz_max_epi64(__mmask8 k, __m256i a, __m256i b);
VPMAXSD __m128i _mm_mask_max_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXSD __m128i _mm_maskz_max_epi32(__mmask8 k, __m128i a, __m128i b);
VPMAXSQ __m128i _mm_mask_max_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXSQ __m128i _mm_maskz_max_epu64(__mmask8 k, __m128i a, __m128i b);
VPMAXSD __m512i _mm512_max_epi32(__m512i a, __m512i b);
VPMAXSD __m512i _mm512_mask_max_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMAXSD __m512i _mm512_maskz_max_epi32(__mmask16 k, __m512i a, __m512i b);
VPMAXSQ __m512i _mm512_max_epi64(__m512i a, __m512i b);
VPMAXSQ __m512i _mm512_mask_max_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMAXSQ __m512i _mm512_maskz_max_epi64(__mmask8 k, __m512i a, __m512i b);
(V)PMAXSB __m128i _mm_max_epi8 (__m128i a, __m128i b);
(V)PMAXSW __m128i _mm_max_epi16 (__m128i a, __m128i b)
(V)PMAXSD __m128i _mm_max_epi32 (__m128i a, __m128i b);
VPMAXSB __m256i _mm256_max_epi8 (__m256i a, __m256i b);
VPMAXSW __m256i _mm256_max_epi16 (__m256i a, __m256i b)
VPMAXSD __m256i _mm256_max_epi32 (__m256i a, __m256i b);
PMAXSW:__m64 _mm_max_pi16(__m64 a, __m64 b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded VPMAXSD/Q, see Table 2-51, "Type E4 Class Exception Conditions."
EVEX-encoded VPMAXSB/W, see Exceptions Type E4.nb in Table 2-51, "Type E4 Class Exception Conditions."

PMAXUB/PMAXUW—Maximum of Packed Unsigned Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DE /r[1] PMAXUB mm1, mm2/m64	A	V/V	SSE	Compare unsigned byte integers in mm2/m64 and mm1 and returns maximum values.
66 0F DE /r PMAXUB xmm1, xmm2/m128	A	V/V	SSE2	Compare packed unsigned byte integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
66 0F 38 3E/r PMAXUW xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1.
VEX.128.66.0F DE /r VPMAXUB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.128.66.0F38 3E/r VPMAXUW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned word integers in xmm3/m128 and xmm2 and store maximum packed values in xmm1.
VEX.256.66.0F DE /r VPMAXUB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
VEX.256.66.0F38 3E/r VPMAXUW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned word integers in ymm3/m256 and ymm2 and store maximum packed values in ymm1.
EVEX.128.66.0F.WIG DE /r VPMAXUB xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.256.66.0F.WIG DE /r VPMAXUB ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.512.66.0F.WIG DE /r VPMAXUB zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Compare packed unsigned byte integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.
EVEX.128.66.0F38.WIG 3E /r VPMAXUW xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed unsigned word integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.256.66.0F38.WIG 3E /r VPMAXUW ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed unsigned word integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.512.66.0F38.WIG 3E /r VPMAXUW zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Compare packed unsigned word integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed unsigned byte, word integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

Legacy SSE version PMAXUB: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMAXUB (64-bit Operands)

```
IF DEST[7:0] > SRC[17:0]) THEN
    DEST[7:0] := DEST[7:0];
ELSE
    DEST[7:0] := SRC[7:0]; FI;
(* Repeat operation for 2nd through 7th bytes in source and destination operands *)
IF DEST[63:56] > SRC[63:56]) THEN
    DEST[63:56] := DEST[63:56];
ELSE
    DEST[63:56] := SRC[63:56]; FI;
```

PMAXUB (128-bit Legacy SSE Version)

```
IF DEST[7:0] >SRC[7:0] THEN
    DEST[7:0] := DEST[7:0];
ELSE
    DEST[15:0] := SRC[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF DEST[127:120] >SRC[127:120] THEN
    DEST[127:120] := DEST[127:120];
ELSE
    DEST[127:120] := SRC[127:120]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXUB (VEX.128 Encoded Version)

```
    IF SRC1[7:0] >SRC2[7:0] THEN
        DEST[7:0] := SRC1[7:0];
    ELSE
        DEST[7:0] := SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF SRC1[127:120] >SRC2[127:120] THEN
        DEST[127:120] := SRC1[127:120];
    ELSE
        DEST[127:120] := SRC2[127:120]; FI;
DEST[MAXVL-1:128] := 0
```

VPMAXUB (VEX.256 Encoded Version)

```
    IF SRC1[7:0] >SRC2[7:0] THEN
        DEST[7:0] := SRC1[7:0];
    ELSE
        DEST[15:0] := SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    IF SRC1[255:248] >SRC2[255:248] THEN
        DEST[255:248] := SRC1[255:248];
    ELSE
        DEST[255:248] := SRC2[255:248]; FI;
DEST[MAXVL-1:128] := 0
```

VPMAXUB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] > SRC2[i+7:i]
            THEN DEST[i+7:i] := SRC1[i+7:i];
            ELSE DEST[i+7:i] := SRC2[i+7:i];
        FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

PMAXUW (128-bit Legacy SSE Version)

```
    IF DEST[15:0] >SRC[15:0] THEN
        DEST[15:0] := DEST[15:0];
    ELSE
        DEST[15:0] := SRC[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:112] >SRC[127:112] THEN
        DEST[127:112] := DEST[127:112];
    ELSE
        DEST[127:112] := SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXUW (VEX.128 Encoded Version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] := SRC1[15:0];
    ELSE
        DEST[15:0] := SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] >SRC2[127:112] THEN
        DEST[127:112] := SRC1[127:112];
    ELSE
        DEST[127:112] := SRC2[127:112]; FI;
DEST[MAXVL-1:128] := 0
```

VPMAXUW (VEX.256 Encoded Version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] := SRC1[15:0];
    ELSE
        DEST[15:0] := SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] >SRC2[255:240] THEN
        DEST[255:240] := SRC1[255:240];
    ELSE
        DEST[255:240] := SRC2[255:240]; FI;
DEST[MAXVL-1:128] := 0
```

VPMAXUW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+15:i] > SRC2[i+15:i]
            THEN DEST[i+15:i] := SRC1[i+15:i];
            ELSE DEST[i+15:i] := SRC2[i+15:i];
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMAXUB __m512i _mm512_max_epu8(__m512i a, __m512i b);
VPMAXUB __m512i _mm512_mask_max_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMAXUB __m512i _mm512_maskz_max_epu8(__mmask64 k, __m512i a, __m512i b);
VPMAXUW __m512i _mm512_max_epu16(__m512i a, __m512i b);
VPMAXUW __m512i _mm512_mask_max_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMAXUW __m512i _mm512_maskz_max_epu16(__mmask32 k, __m512i a, __m512i b);
VPMAXUB __m256i _mm256_mask_max_epu8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMAXUB __m256i _mm256_maskz_max_epu8(__mmask32 k, __m256i a, __m256i b);
VPMAXUW __m256i _mm256_mask_max_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXUW __m256i _mm256_maskz_max_epu16(__mmask16 k, __m256i a, __m256i b);
VPMAXUB __m128i _mm_mask_max_epu8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMAXUB __m128i _mm_maskz_max_epu8(__mmask16 k, __m128i a, __m128i b);
VPMAXUW __m128i _mm_mask_max_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXUW __m128i _mm_maskz_max_epu16(__mmask8 k, __m128i a, __m128i b);
(V)PMAXUB __m128i _mm_max_epu8 (__m128i a, __m128i b);
(V)PMAXUW __m128i _mm_max_epu16 (__m128i a, __m128i b)
VPMAXUB __m256i _mm256_max_epu8 (__m256i a, __m256i b);
VPMAXUW __m256i _mm256_max_epu16 (__m256i a, __m256i b);
PMAXUB __m64 _mm_max_pu8(__m64 a, __m64 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

PMAXUD/PMAXUQ—Maximum of Packed Unsigned Integers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3F /r PMAXUD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned dword integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
VEX.128.66.0F38.WIG 3F /r VPMAXUD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.256.66.0F38.WIG 3F /r VPMAXUD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned dword integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
EVEX.128.66.0F38.W0 3F /r VPMAXUD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed unsigned dword integers in xmm2 and xmm3/m128/m32bcst and store packed maximum values in xmm1 under writemask k1.
EVEX.256.66.0F38.W0 3F /r VPMAXUD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed unsigned dword integers in ymm2 and ymm3/m256/m32bcst and store packed maximum values in ymm1 under writemask k1.
EVEX.512.66.0F38.W0 3F /r VPMAXUD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Compare packed unsigned dword integers in zmm2 and zmm3/m512/m32bcst and store packed maximum values in zmm1 under writemask k1.
EVEX.128.66.0F38.W1 3F /r VPMAXUQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed unsigned qword integers in xmm2 and xmm3/m128/m64bcst and store packed maximum values in xmm1 under writemask k1.
EVEX.256.66.0F38.W1 3F /r VPMAXUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed unsigned qword integers in ymm2 and ymm3/m256/m64bcst and store packed maximum values in ymm1 under writemask k1.
EVEX.512.66.0F38.W1 3F /r VPMAXUQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Compare packed unsigned qword integers in zmm2 and zmm3/m512/m64bcst and store packed maximum values in zmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed unsigned dword or qword integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register; The second source operand is a YMM register or 256-bit memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMAXUD (128-bit Legacy SSE Version)

```
    IF DEST[31:0] >SRC[31:0] THEN
        DEST[31:0] := DEST[31:0];
    ELSE
        DEST[31:0] := SRC[31:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:96] >SRC[127:96] THEN
        DEST[127:96] := DEST[127:96];
    ELSE
        DEST[127:96] := SRC[127:96]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXUD (VEX.128 Encoded Version)

```
    IF SRC1[31:0] > SRC2[31:0] THEN
        DEST[31:0] := SRC1[31:0];
    ELSE
        DEST[31:0] := SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
    IF SRC1[127:96] > SRC2[127:96] THEN
        DEST[127:96] := SRC1[127:96];
    ELSE
        DEST[127:96] := SRC2[127:96]; FI;
DEST[MAXVL-1:128] := 0
```

VPMAXUD (VEX.256 Encoded Version)

```
    IF SRC1[31:0] > SRC2[31:0] THEN
        DEST[31:0] := SRC1[31:0];
    ELSE
        DEST[31:0] := SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 7th dwords in source and destination operands *)
    IF SRC1[255:224] > SRC2[255:224] THEN
        DEST[255:224] := SRC1[255:224];
    ELSE
        DEST[255:224] := SRC2[255:224]; FI;
DEST[MAXVL-1:256] := 0
```

VPMAXUD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+31:i] > SRC2[31:0]
                    THEN DEST[i+31:i] := SRC1[i+31:i];
                    ELSE DEST[i+31:i] := SRC2[31:0];
                FI;
            ELSE
                IF SRC1[i+31:i] > SRC2[i+31:i]
                    THEN DEST[i+31:i] := SRC1[i+31:i];
                    ELSE DEST[i+31:i] := SRC2[i+31:i];
            FI;
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    THEN DEST[i+31:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPMAXUQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+63:i] > SRC2[63:0]
                    THEN DEST[i+63:i] := SRC1[i+63:i];
                    ELSE DEST[i+63:i] := SRC2[63:0];
                FI;
            ELSE
                IF SRC1[i+31:i] > SRC2[i+31:i]
                    THEN DEST[i+63:i] := SRC1[i+63:i];
                    ELSE DEST[i+63:i] := SRC2[i+63:i];
            FI;
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    THEN DEST[i+63:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMAXUD __m512i _mm512_max_epu32(__m512i a, __m512i b);
VPMAXUD __m512i _mm512_mask_max_epu32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMAXUD __m512i _mm512_maskz_max_epu32(__mmask16 k, __m512i a, __m512i b);
VPMAXUQ __m512i _mm512_max_epu64(__m512i a, __m512i b);
VPMAXUQ __m512i _mm512_mask_max_epu64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMAXUQ __m512i _mm512_maskz_max_epu64(__mmask8 k, __m512i a, __m512i b);
VPMAXUD __m256i _mm256_mask_max_epu32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXUD __m256i _mm256_maskz_max_epu32(__mmask16 k, __m256i a, __m256i b);
VPMAXUQ __m256i _mm256_mask_max_epu64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMAXUQ __m256i _mm256_maskz_max_epu64(__mmask8 k, __m256i a, __m256i b);
VPMAXUD __m128i _mm_mask_max_epu32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXUD __m128i _mm_maskz_max_epu32(__mmask8 k, __m128i a, __m128i b);
VPMAXUQ __m128i _mm_mask_max_epu64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXUQ __m128i _mm_maskz_max_epu64(__mmask8 k, __m128i a, __m128i b);
(V)PMAXUD __m128i _mm_max_epu32 (__m128i a, __m128i b);
VPMAXUD __m256i _mm256_max_epu32 (__m256i a, __m256i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

PMINSB/PMINSW—Minimum of Packed Signed Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EA /r[1] PMINSW mm1, mm2/m64	A	V/V	SSE	Compare signed word integers in mm2/m64 and mm1 and return minimum values.
66 0F 38 38 /r PMINSB xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
66 0F EA /r PMINSW xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1.
VEX.128.66.0F38 38 /r VPMINSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.128.66.0F EA /r VPMINSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1.
VEX.256.66.0F38 38 /r VPMINSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.
VEX.256.66.0F EA /r VPMINSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1.
EVEX.128.66.0F38.WIG 38 /r VPMINSB xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F38.WIG 38 /r VPMINSB ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F38.WIG 38 /r VPMINSB zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Compare packed signed byte integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.
EVEX.128.66.0F.WIG EA /r VPMINSW xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed word integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F.WIG EA /r VPMINSW ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed word integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F.WIG EA /r VPMINSW zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Compare packed signed word integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed signed byte, word, or dword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

Legacy SSE version PMINSW: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMINSW (64-bit Operands)

```
IF DEST[15:0] < SRC[15:0] THEN
    DEST[15:0] := DEST[15:0];
ELSE
    DEST[15:0] := SRC[15:0]; FI;
(* Repeat operation for 2nd and 3rd words in source and destination operands *)
IF DEST[63:48] < SRC[63:48] THEN
    DEST[63:48] := DEST[63:48];
ELSE
    DEST[63:48] := SRC[63:48]; FI;
```

PMINSB (128-bit Legacy SSE Version)

```
IF DEST[7:0] < SRC[7:0] THEN
    DEST[7:0] := DEST[7:0];
ELSE
    DEST[15:0] := SRC[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF DEST[127:120] < SRC[127:120] THEN
    DEST[127:120] := DEST[127:120];
ELSE
    DEST[127:120] := SRC[127:120]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINSB (VEX.128 Encoded Version)

```
    IF SRC1[7:0] < SRC2[7:0] THEN
        DEST[7:0] := SRC1[7:0];
    ELSE
        DEST[7:0] := SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF SRC1[127:120] < SRC2[127:120] THEN
        DEST[127:120] := SRC1[127:120];
    ELSE
        DEST[127:120] := SRC2[127:120]; FI;
DEST[MAXVL-1:128] := 0
```

VPMINSB (VEX.256 Encoded Version)

```
    IF SRC1[7:0] < SRC2[7:0] THEN
        DEST[7:0] := SRC1[7:0];
    ELSE
        DEST[15:0] := SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    IF SRC1[255:248] < SRC2[255:248] THEN
        DEST[255:248] := SRC1[255:248];
    ELSE
        DEST[255:248] := SRC2[255:248]; FI;
DEST[MAXVL-1:256] := 0
```

VPMINSB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] < SRC2[i+7:i]
            THEN DEST[i+7:i] := SRC1[i+7:i];
            ELSE DEST[i+7:i] := SRC2[i+7:i];
        FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

PMINSW (128-bit Legacy SSE Version)

```
    IF DEST[15:0] < SRC[15:0] THEN
        DEST[15:0] := DEST[15:0];
    ELSE
        DEST[15:0] := SRC[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:112] < SRC[127:112] THEN
        DEST[127:112] := DEST[127:112];
    ELSE
        DEST[127:112] := SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINSW (VEX.128 Encoded Version)

```
    IF SRC1[15:0] < SRC2[15:0] THEN
        DEST[15:0] := SRC1[15:0];
    ELSE
        DEST[15:0] := SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] < SRC2[127:112] THEN
        DEST[127:112] := SRC1[127:112];
    ELSE
        DEST[127:112] := SRC2[127:112]; FI;
DEST[MAXVL-1:128] := 0
```

VPMINSW (VEX.256 Encoded Version)

```
    IF SRC1[15:0] < SRC2[15:0] THEN
        DEST[15:0] := SRC1[15:0];
    ELSE
        DEST[15:0] := SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] < SRC2[255:240] THEN
        DEST[255:240] := SRC1[255:240];
    ELSE
        DEST[255:240] := SRC2[255:240]; FI;
DEST[MAXVL-1:256] := 0
```

VPMINSW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+15:i] < SRC2[i+15:i]
            THEN DEST[i+15:i] := SRC1[i+15:i];
            ELSE DEST[i+15:i] := SRC2[i+15:i];
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMINSB __m512i _mm512_min_epi8(__m512i a, __m512i b);
VPMINSB __m512i _mm512_mask_min_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMINSB __m512i _mm512_maskz_min_epi8(__mmask64 k, __m512i a, __m512i b);
VPMINSW __m512i _mm512_min_epi16(__m512i a, __m512i b);
VPMINSW __m512i _mm512_mask_min_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMINSW __m512i _mm512_maskz_min_epi16(__mmask32 k, __m512i a, __m512i b);
VPMINSB __m256i _mm256_mask_min_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMINSB __m256i _mm256_maskz_min_epi8(__mmask32 k, __m256i a, __m256i b);
VPMINSW __m256i _mm256_mask_min_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINSW __m256i _mm256_maskz_min_epi16(__mmask16 k, __m256i a, __m256i b);
VPMINSB __m128i _mm_mask_min_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMINSB __m128i _mm_maskz_min_epi8(__mmask16 k, __m128i a, __m128i b);
VPMINSW __m128i _mm_mask_min_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINSW __m128i _mm_maskz_min_epi16(__mmask8 k, __m128i a, __m128i b);
(V)PMINSB __m128i _mm_min_epi8 (__m128i a, __m128i b);
(V)PMINSW __m128i _mm_min_epi16 (__m128i a, __m128i b)
VPMINSB __m256i _mm256_min_epi8 (__m256i a, __m256i b);
VPMINSW __m256i _mm256_min_epi16 (__m256i a, __m256i b)
PMINSW__m64 _mm_min_pi16 (__m64 a, __m64 b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”
Additionally:

#MF	(64-bit operations only) If there is a pending x87 FPU exception.

PMINSD/PMINSQ—Minimum of Packed Signed Integers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 39 /r PMINSD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed dword integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
VEX.128.66.0F38.WIG 39 /r VPMINSD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.256.66.0F38.WIG 39 /r VPMINSD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed dword integers in ymm2 and ymm3/m128 and store packed minimum values in ymm1.
EVEX.128.66.0F38.W0 39 /r VPMINSD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F38.W0 39 /r VPMINSD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed signed dword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F38.W0 39 /r VPMINSD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Compare packed signed dword integers in zmm2 and zmm3/m512/m32bcst and store packed minimum values in zmm1 under writemask k1.
EVEX.128.66.0F38.W1 39 /r VPMINSQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed signed qword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F38.W1 39 /r VPMINSQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed signed qword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F38.W1 39 /r VPMINSQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Compare packed signed qword integers in zmm2 and zmm3/m512/m64bcst and store packed minimum values in zmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed signed dword or qword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMINSD (128-bit Legacy SSE Version)

```
    IF DEST[31:0] < SRC[31:0] THEN
        DEST[31:0] := DEST[31:0];
    ELSE
        DEST[31:0] := SRC[31:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:96] < SRC[127:96] THEN
        DEST[127:96] := DEST[127:96];
    ELSE
        DEST[127:96] := SRC[127:96]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINSD (VEX.128 Encoded Version)

```
    IF SRC1[31:0] < SRC2[31:0] THEN
        DEST[31:0] := SRC1[31:0];
    ELSE
        DEST[31:0] := SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
    IF SRC1[127:96] < SRC2[127:96] THEN
        DEST[127:96] := SRC1[127:96];
    ELSE
        DEST[127:96] := SRC2[127:96]; FI;
DEST[MAXVL-1:128] := 0
```

VPMINSD (VEX.256 Encoded Version)

```
    IF SRC1[31:0] < SRC2[31:0] THEN
        DEST[31:0] := SRC1[31:0];
    ELSE
        DEST[31:0] := SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 7th dwords in source and destination operands *)
    IF SRC1[255:224] < SRC2[255:224] THEN
        DEST[255:224] := SRC1[255:224];
    ELSE
        DEST[255:224] := SRC2[255:224]; FI;
DEST[MAXVL-1:256] := 0
```

VPMINSD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+31:i] < SRC2[31:0]
                    THEN DEST[i+31:i] := SRC1[i+31:i];
                    ELSE DEST[i+31:i] := SRC2[31:0];
                FI;
            ELSE
                IF SRC1[i+31:i] < SRC2[i+31:i]
                    THEN DEST[i+31:i] := SRC1[i+31:i];
                    ELSE DEST[i+31:i] := SRC2[i+31:i];
            FI;
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPMINSQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+63:i] < SRC2[63:0]
                    THEN DEST[i+63:i] := SRC1[i+63:i];
                    ELSE DEST[i+63:i] := SRC2[63:0];
                FI;
            ELSE
                IF SRC1[i+63:i] < SRC2[i+63:i]
                    THEN DEST[i+63:i] := SRC1[i+63:i];
                    ELSE DEST[i+63:i] := SRC2[i+63:i];
            FI;
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMINSD __m512i _mm512_min_epi32(__m512i a, __m512i b);
VPMINSD __m512i _mm512_mask_min_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMINSD __m512i _mm512_maskz_min_epi32(__mmask16 k, __m512i a, __m512i b);
VPMINSQ __m512i _mm512_min_epi64(__m512i a, __m512i b);
VPMINSQ __m512i _mm512_mask_min_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMINSQ __m512i _mm512_maskz_min_epi64(__mmask8 k, __m512i a, __m512i b);
VPMINSD __m256i _mm256_mask_min_epi32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINSD __m256i _mm256_maskz_min_epi32(__mmask16 k, __m256i a, __m256i b);
VPMINSQ __m256i _mm256_mask_min_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMINSQ __m256i _mm256_maskz_min_epi64(__mmask8 k, __m256i a, __m256i b);
VPMINSD __m128i _mm_mask_min_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINSD __m128i _mm_maskz_min_epi32(__mmask8 k, __m128i a, __m128i b);
VPMINSQ __m128i _mm_mask_min_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINSQ __m128i _mm_maskz_min_epu64(__mmask8 k, __m128i a, __m128i b);
(V)PMINSD __m128i _mm_min_epi32 (__m128i a, __m128i b);
VPMINSD __m256i _mm256_min_epi32 (__m256i a, __m256i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

PMINUB/PMINUW—Minimum of Packed Unsigned Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DA /r[1] PMINUB mm1, mm2/m64	A	V/V	SSE	Compare unsigned byte integers in mm2/m64 and mm1 and returns minimum values.
66 0F DA /r PMINUB xmm1, xmm2/m128	A	V/V	SSE2	Compare packed unsigned byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
66 0F 38 3A/r PMINUW xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1.
VEX.128.66.0F DA /r VPMINUB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.128.66.0F38 3A/r VPMINUW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1.
VEX.256.66.0F DA /r VPMINUB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.
VEX.256.66.0F38 3A/r VPMINUW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1.
EVEX.128.66.0F DA /r VPMINUB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F DA /r VPMINUB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F DA /r VPMINUB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Compare packed unsigned byte integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.
EVEX.128.66.0F38 3A/r VPMINUW xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed unsigned word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F38 3A/r VPMINUW ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed unsigned word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F38 3A/r VPMINUW zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Compare packed unsigned word integers in zmm3/m512 and zmm2 and return packed minimum values in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed unsigned byte or word integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

Legacy SSE version PMINUB: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMINUB (64-bit Operands)

```
IF DEST[7:0] < SRC[17:0] THEN
    DEST[7:0] := DEST[7:0];
ELSE
    DEST[7:0] := SRC[7:0]; FI;
(* Repeat operation for 2nd through 7th bytes in source and destination operands *)
IF DEST[63:56] < SRC[63:56] THEN
    DEST[63:56] := DEST[63:56];
ELSE
    DEST[63:56] := SRC[63:56]; FI;
```

PMINUB (128-bit Operands)

```
IF DEST[7:0] < SRC[7:0] THEN
    DEST[7:0] := DEST[7:0];
ELSE
    DEST[15:0] := SRC[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF DEST[127:120] < SRC[127:120] THEN
    DEST[127:120] := DEST[127:120];
ELSE
    DEST[127:120] := SRC[127:120]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINUB (VEX.128 Encoded Version)

```
    IF SRC1[7:0] < SRC2[7:0] THEN
        DEST[7:0] := SRC1[7:0];
    ELSE
        DEST[7:0] := SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF SRC1[127:120] < SRC2[127:120] THEN
        DEST[127:120] := SRC1[127:120];
    ELSE
        DEST[127:120] := SRC2[127:120]; FI;
DEST[MAXVL-1:128] := 0
```

VPMINUB (VEX.256 Encoded Version)

```
    IF SRC1[7:0] < SRC2[7:0] THEN
        DEST[7:0] := SRC1[7:0];
    ELSE
        DEST[15:0] := SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    IF SRC1[255:248] < SRC2[255:248] THEN
        DEST[255:248] := SRC1[255:248];
    ELSE
        DEST[255:248] := SRC2[255:248]; FI;
DEST[MAXVL-1:256] := 0
```

VPMINUB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] < SRC2[i+7:i]
            THEN DEST[i+7:i] := SRC1[i+7:i];
            ELSE DEST[i+7:i] := SRC2[i+7:i];
        FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

PMINUW (128-bit Operands)

```
    IF DEST[15:0] < SRC[15:0] THEN
        DEST[15:0] := DEST[15:0];
    ELSE
        DEST[15:0] := SRC[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:112] < SRC[127:112] THEN
        DEST[127:112] := DEST[127:112];
    ELSE
        DEST[127:112] := SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINUW (VEX.128 Encoded Version)

```
    IF SRC1[15:0] < SRC2[15:0] THEN
        DEST[15:0] := SRC1[15:0];
    ELSE
        DEST[15:0] := SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] < SRC2[127:112] THEN
        DEST[127:112] := SRC1[127:112];
    ELSE
        DEST[127:112] := SRC2[127:112]; FI;
DEST[MAXVL-1:128] := 0
```

VPMINUW (VEX.256 Encoded Version)

```
    IF SRC1[15:0] < SRC2[15:0] THEN
        DEST[15:0] := SRC1[15:0];
    ELSE
        DEST[15:0] := SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] < SRC2[255:240] THEN
        DEST[255:240] := SRC1[255:240];
    ELSE
        DEST[255:240] := SRC2[255:240]; FI;
DEST[MAXVL-1:256] := 0
```

VPMINUW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+15:i] < SRC2[i+15:i]
            THEN DEST[i+15:i] := SRC1[i+15:i];
            ELSE DEST[i+15:i] := SRC2[i+15:i];
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMINUB __m512i _mm512_min_epu8(__m512i a, __m512i b);
VPMINUB __m512i _mm512_mask_min_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMINUB __m512i _mm512_maskz_min_epu8(__mmask64 k, __m512i a, __m512i b);
VPMINUW __m512i _mm512_min_epu16(__m512i a, __m512i b);
VPMINUW __m512i _mm512_mask_min_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMINUW __m512i _mm512_maskz_min_epu16(__mmask32 k, __m512i a, __m512i b);
VPMINUB __m256i _mm256_mask_min_epu8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMINUB __m256i _mm256_maskz_min_epu8(__mmask32 k, __m256i a, __m256i b);
VPMINUW __m256i _mm256_mask_min_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINUW __m256i _mm256_maskz_min_epu16(__mmask16 k, __m256i a, __m256i b);
VPMINUB __m128i _mm_mask_min_epu8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMINUB __m128i _mm_maskz_min_epu8(__mmask16 k, __m128i a, __m128i b);
VPMINUW __m128i _mm_mask_min_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINUW __m128i _mm_maskz_min_epu16(__mmask8 k, __m128i a, __m128i b);
(V)PMINUB __m128i _mm_min_epu8 (__m128i a, __m128i b)
(V)PMINUW __m128i _mm_min_epu16 (__m128i a, __m128i b);
VPMINUB __m256i _mm256_min_epu8 (__m256i a, __m256i b)
VPMINUW __m256i _mm256_min_epu16 (__m256i a, __m256i b);
PMINUB __m64 _m_min_pu8 (__m64 a, __m64 b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

PMINUD/PMINUQ—Minimum of Packed Unsigned Integers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3B /r PMINUD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned dword integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
VEX.128.66.0F38.WIG 3B /r VPMINUD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.256.66.0F38.WIG 3B /r VPMINUD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned dword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.
EVEX.128.66.0F38.W0 3B /r VPMINUD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed unsigned dword integers in xmm2 and xmm3/m128/m32bcst and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F38.W0 3B /r VPMINUD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed unsigned dword integers in ymm2 and ymm3/m256/m32bcst and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F38.W0 3B /r VPMINUD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Compare packed unsigned dword integers in zmm2 and zmm3/m512/m32bcst and store packed minimum values in zmm1 under writemask k1.
EVEX.128.66.0F38.W1 3B /r VPMINUQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed unsigned qword integers in xmm2 and xmm3/m128/m64bcst and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F38.W1 3B /r VPMINUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed unsigned qword integers in ymm2 and ymm3/m256/m64bcst and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F38.W1 3B /r VPMINUQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Compare packed unsigned qword integers in zmm2 and zmm3/m512/m64bcst and store packed minimum values in zmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed unsigned dword/qword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMINUD (128-bit Legacy SSE Version)

```
PMINUD instruction for 128-bit operands:
    IF DEST[31:0] < SRC[31:0] THEN
        DEST[31:0] := DEST[31:0];
    ELSE
        DEST[31:0] := SRC[31:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:96] < SRC[127:96] THEN
        DEST[127:96] := DEST[127:96];
    ELSE
        DEST[127:96] := SRC[127:96]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINUD (VEX.128 Encoded Version)

```
VPMINUD instruction for 128-bit operands:
    IF SRC1[31:0] < SRC2[31:0] THEN
        DEST[31:0] := SRC1[31:0];
    ELSE
        DEST[31:0] := SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
    IF SRC1[127:96] < SRC2[127:96] THEN
        DEST[127:96] := SRC1[127:96];
    ELSE
        DEST[127:96] := SRC2[127:96]; FI;
DEST[MAXVL-1:128] := 0
```

VPMINUD (VEX.256 Encoded Version)

```
VPMINUD instruction for 128-bit operands:
    IF SRC1[31:0] < SRC2[31:0] THEN
        DEST[31:0] := SRC1[31:0];
    ELSE
        DEST[31:0] := SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 7th dwords in source and destination operands *)
    IF SRC1[255:224] < SRC2[255:224] THEN
        DEST[255:224] := SRC1[255:224];
    ELSE
        DEST[255:224] := SRC2[255:224]; FI;
DEST[MAXVL-1:256] := 0
```

VPMINUD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+31:i] < SRC2[31:0]
                    THEN DEST[i+31:i] := SRC1[i+31:i];
                    ELSE DEST[i+31:i] := SRC2[31:0];
                FI;
            ELSE
                IF SRC1[i+31:i] < SRC2[i+31:i]
                    THEN DEST[i+31:i] := SRC1[i+31:i];
                    ELSE DEST[i+31:i] := SRC2[i+31:i];
            FI;
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPMINUQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+63:i] < SRC2[63:0]
                    THEN DEST[i+63:i] := SRC1[i+63:i];
                    ELSE DEST[i+63:i] := SRC2[63:0];
                FI;
            ELSE
                IF SRC1[i+63:i] < SRC2[i+63:i]
                    THEN DEST[i+63:i] := SRC1[i+63:i];
                    ELSE DEST[i+63:i] := SRC2[i+63:i];
            FI;
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMINUD __m512i _mm512_min_epu32(__m512i a, __m512i b);
VPMINUD __m512i _mm512_mask_min_epu32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMINUD __m512i _mm512_maskz_min_epu32(__mmask16 k, __m512i a, __m512i b);
VPMINUQ __m512i _mm512_min_epu64(__m512i a, __m512i b);
VPMINUQ __m512i _mm512_mask_min_epu64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMINUQ __m512i _mm512_maskz_min_epu64(__mmask8 k, __m512i a, __m512i b);
VPMINUD __m256i _mm256_mask_min_epu32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINUD __m256i _mm256_maskz_min_epu32(__mmask16 k, __m256i a, __m256i b);
VPMINUQ __m256i _mm256_mask_min_epu64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMINUQ __m256i _mm256_maskz_min_epu64(__mmask8 k, __m256i a, __m256i b);
VPMINUD __m128i _mm_mask_min_epu32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINUD __m128i _mm_maskz_min_epu32(__mmask8 k, __m128i a, __m128i b);
VPMINUQ __m128i _mm_mask_min_epu64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINUQ __m128i _mm_maskz_min_epu64(__mmask8 k, __m128i a, __m128i b);
(V)PMINUD __m128i _mm_min_epu32 (__m128i a, __m128i b);
VPMINUD __m256i _mm256_min_epu32 (__m256i a, __m256i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

PMOVMSKB—Move Byte Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F D7 /r[1] PMOVMSKB reg, mm	RM	V/V	SSE	Move a byte mask of mm to reg. The upper bits of r32 or r64 are zeroed
66 0F D7 /r PMOVMSKB reg, xmm	RM	V/V	SSE2	Move a byte mask of xmm to reg. The upper bits of r32 or r64 are zeroed
VEX.128.66.0F.WIG D7 /r VPMOVMSKB reg, xmm1	RM	V/V	AVX	Move a byte mask of xmm1 to reg. The upper bits of r32 or r64 are filled with zeros.
VEX.256.66.0F.WIG D7 /r VPMOVMSKB reg, ymm1	RM	V/V	AVX2	Move a 32-bit mask of ymm1 to reg. The upper bits of r64 are filled with zeros.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Creates a mask made up of the most significant bit of each byte of the source operand (second operand) and stores the result in the low byte or word of the destination operand (first operand).

The byte mask is 8 bits for 64-bit source operand, 16 bits for 128-bit source operand and 32 bits for 256-bit source operand. The destination operand is a general-purpose register.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

Legacy SSE version: The source operand is an MMX technology register.

128-bit Legacy SSE version: The source operand is an XMM register.

VEX.128 encoded version: The source operand is an XMM register.

VEX.256 encoded version: The source operand is a YMM register.

Note: VEX.vvvv is reserved and must be 1111b.

Operation

PMOVMSKB (With 64-bit Source Operand and r32)

```
r32[0] := SRC[7];
r32[1] := SRC[15];
(* Repeat operation for bytes 2 through 6 *)
r32[7] := SRC[63];
r32[31:8] := ZERO_FILL;
```

(V)PMOVMSKB (With 128-bit Source Operand and r32)

```
r32[0] := SRC[7];
r32[1] := SRC[15];
(* Repeat operation for bytes 2 through 14 *)
r32[15] := SRC[127];
r32[31:16] := ZERO_FILL;
```

VPMOVMSKB (With 256-bit Source Operand and r32)

```
r32[0] := SRC[7];
r32[1] := SRC[15];
(* Repeat operation for bytes 3rd through 31*)
r32[31] := SRC[255];
```

PMOVMSKB (With 64-bit Source Operand and r64)

```
r64[0] := SRC[7];
r64[1] := SRC[15];
(* Repeat operation for bytes 2 through 6 *)
r64[7] := SRC[63];
r64[63:8] := ZERO_FILL;
```

(V)PMOVMSKB (With 128-bit Source Operand and r64)

```
r64[0] := SRC[7];
r64[1] := SRC[15];
(* Repeat operation for bytes 2 through 14 *)
r64[15] := SRC[127];
r64[63:16] := ZERO_FILL;
```

VPMOVMSKB (With 256-bit Source Operand and r64)

```
r64[0] := SRC[7];
r64[1] := SRC[15];
(* Repeat operation for bytes 2 through 31*)
r64[31] := SRC[255];
r64[63:32] := ZERO_FILL;
```

Intel C/C++ Compiler Intrinsic Equivalent

PMOVMSKB int _mm_movemask_pi8(__m64 a)
(V)PMOVMSKB int _mm_movemask_epi8 (__m128i a)
VPMOVMSKB int _mm256_movemask_epi8 (__m256i a)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Table 2-24, "Type 7 Class Exception Conditions," additionally:

#UD	If VEX.vvvv ≠ 1111B.

PMOVSX—Packed Move With Sign Extend

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0f 38 20 /r PMOVSXBW xmm1, xmm2/m64	A	V/V	SSE4_1	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
66 0f 38 21 /r PMOVSXBD xmm1, xmm2/m32	A	V/V	SSE4_1	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
66 0f 38 22 /r PMOVSXBQ xmm1, xmm2/m16	A	V/V	SSE4_1	Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
66 0f 38 23/r PMOVSXWD xmm1, xmm2/m64	A	V/V	SSE4_1	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
66 0f 38 24 /r PMOVSXWQ xmm1, xmm2/m32	A	V/V	SSE4_1	Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
66 0f 38 25 /r PMOVSXDQ xmm1, xmm2/m64	A	V/V	SSE4_1	Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 20 /r VPMOVSXBW xmm1, xmm2/m64	A	V/V	AVX	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
VEX.128.66.0F38.WIG 21 /r VPMOVSXBD xmm1, xmm2/m32	A	V/V	AVX	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 22 /r VPMOVSXBQ xmm1, xmm2/m16	A	V/V	AVX	Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 23 /r VPMOVSXWD xmm1, xmm2/m64	A	V/V	AVX	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 24 /r VPMOVSXWQ xmm1, xmm2/m32	A	V/V	AVX	Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 25 /r VPMOVSXDQ xmm1, xmm2/m64	A	V/V	AVX	Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.256.66.0F38.WIG 20 /r VPMOVSXBW ymm1, xmm2/m128	A	V/V	AVX2	Sign extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.
VEX.256.66.0F38.WIG 21 /r VPMOVSXBD ymm1, xmm2/m64	A	V/V	AVX2	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1.
VEX.256.66.0F38.WIG 22 /r VPMOVSXBQ ymm1, xmm2/m32	A	V/V	AVX2	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1.
VEX.256.66.0F38.WIG 23 /r VPMOVSXWD ymm1, xmm2/m128	A	V/V	AVX2	Sign extend 8 packed 16-bit integers in xmm2/m128 to 8 packed 32-bit integers in ymm1.
VEX.256.66.0F38.WIG 24 /r VPMOVSXWQ ymm1, xmm2/m64	A	V/V	AVX2	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1.
VEX.256.66.0F38.WIG 25 /r VPMOVSXDQ ymm1, xmm2/m128	A	V/V	AVX2	Sign extend 4 packed 32-bit integers in xmm2/m128 to 4 packed 64-bit integers in ymm1.
EVEX.128.66.0F38.WIG 20 /r VPMOVSXBW xmm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F38.WIG 20 /r VPMOVSXBW ymm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Sign extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 20 /r VPMOVSXBW zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512BW OR AVX10.1	Sign extend 32 packed 8-bit integers in ymm2/m256 to 32 packed 16-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 21 /r VPMOVSXBD xmm1 {k1}{z}, xmm2/m32	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 21 /r VPMOVSXBD ymm1 {k1}{z}, xmm2/m64	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 21 /r VPMOVSXBD zmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512F OR AVX10.1	Sign extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 22 /r VPMOVSXBQ xmm1 {k1}{z}, xmm2/m16	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 22 /r VPMOVSXBQ ymm1 {k1}{z}, xmm2/m32	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 22 /r VPMOVSXBQ zmm1 {k1}{z}, xmm2/m64	D	V/V	AVX512F OR AVX10.1	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 23 /r VPMOVSXWD xmm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 23 /r VPMOVSXWD ymm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Sign extend 8 packed 16-bit integers in xmm2/m128 to 8 packed 32-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 23 /r VPMOVSXWD zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512F OR AVX10.1	Sign extend 16 packed 16-bit integers in ymm2/m256 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 24 /r VPMOVSXWQ xmm1 {k1}{z}, xmm2/m32	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 24 /r VPMOVSXWQ ymm1 {k1}{z}, xmm2/m64	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 24 /r VPMOVSXWQ zmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512F OR AVX10.1	Sign extend 8 packed 16-bit integers in xmm2/m128 to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.W0 25 /r VPMOVSXDQ xmm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1 using writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F38.W0 25 /r VPMOVSXDQ ymm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Sign extend 4 packed 32-bit integers in xmm2/m128 to 4 packed 64-bit integers in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 25 /r VPMOVSXDQ zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512F OR AVX10.1	Sign extend 8 packed 32-bit integers in ymm2/m256 to 8 packed 64-bit integers in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Half Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
C	Quarter Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
D	Eighth Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Legacy and VEX encoded versions: Packed byte, word, or dword integers starting from the low bytes of the source operand (second operand) are sign extended to packed word, dword, or quadword integers in the destination operand.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: Packed byte, word, or dword integers starting from the low bytes of the source operand (second operand) are sign extended to packed word, dword, or quadword integers in the destination operand under the writemask.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

Packed_Sign_Extend_BYTE_to_WORD(DEST, SRC)

```
DEST[15:0] := SignExtend(SRC[7:0]);
DEST[31:16] := SignExtend(SRC[15:8]);
DEST[47:32] := SignExtend(SRC[23:16]);
DEST[63:48] := SignExtend(SRC[31:24]);
DEST[79:64] := SignExtend(SRC[39:32]);
DEST[95:80] := SignExtend(SRC[47:40]);
DEST[111:96] := SignExtend(SRC[55:48]);
DEST[127:112] := SignExtend(SRC[63:56]);
```

Packed_Sign_Extend_BYTE_to_DWORD(DEST, SRC)
```
DEST[31:0] := SignExtend(SRC[7:0]);
DEST[63:32] := SignExtend(SRC[15:8]);
DEST[95:64] := SignExtend(SRC[23:16]);
DEST[127:96] := SignExtend(SRC[31:24]);
```

Packed_Sign_Extend_BYTE_to_QWORD(DEST, SRC)
```
DEST[63:0] := SignExtend(SRC[7:0]);
DEST[127:64] := SignExtend(SRC[15:8]);
```

Packed_Sign_Extend_WORD_to_DWORD(DEST, SRC)
```
DEST[31:0] := SignExtend(SRC[15:0]);
DEST[63:32] := SignExtend(SRC[31:16]);
DEST[95:64] := SignExtend(SRC[47:32]);
DEST[127:96] := SignExtend(SRC[63:48]);
```

Packed_Sign_Extend_WORD_to_QWORD(DEST, SRC)
```
DEST[63:0] := SignExtend(SRC[15:0]);
DEST[127:64] := SignExtend(SRC[31:16]);
```

Packed_Sign_Extend_DWORD_to_QWORD(DEST, SRC)
```
DEST[63:0] := SignExtend(SRC[31:0]);
DEST[127:64] := SignExtend(SRC[63:32]);
```

VPMOVSXBW (EVEX Encoded Versions)
```
(KL, VL) = (8, 128), (16, 256), (32, 512)
Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[383:256], SRC[191:128])
    Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[511:384], SRC[255:192])
FI;
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TEMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVSXBD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
    Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
    Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[383:256], SRC[95:64])
    Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TEMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVSXBQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[127:0], SRC[15:0])
IF VL >= 256
    Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[255:128], SRC[31:16])
FI;
IF VL >= 512
    Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[383:256], SRC[47:32])
    Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[511:384], SRC[63:48])
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVSXWD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Sign_Extend_WORD_to_DWORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Sign_Extend_WORD_to_DWORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Sign_Extend_WORD_to_DWORD(TMP_DEST[383:256], SRC[191:128])
    Packed_Sign_Extend_WORD_to_DWORD(TMP_DEST[511:384], SRC[256:192])
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TEMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVSXWQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Sign_Extend_WORD_to_QWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
    Packed_Sign_Extend_WORD_to_QWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
    Packed_Sign_Extend_WORD_to_QWORD(TMP_DEST[383:256], SRC[95:64])
    Packed_Sign_Extend_WORD_to_QWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVSXDQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Sign_Extend_DWORD_to_QWORD(TEMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Sign_Extend_DWORD_to_QWORD(TEMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Sign_Extend_DWORD_to_QWORD(TEMP_DEST[383:256], SRC[191:128])
    Packed_Sign_Extend_DWORD_to_QWORD(TEMP_DEST[511:384], SRC[255:192])
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVSXBW (VEX.256 Encoded Version)

```
Packed_Sign_Extend_BYTE_to_WORD(DEST[127:0], SRC[63:0])
Packed_Sign_Extend_BYTE_to_WORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] := 0
```

VPMOVSXBD (VEX.256 Encoded Version)

```
Packed_Sign_Extend_BYTE_to_DWORD(DEST[127:0], SRC[31:0])
Packed_Sign_Extend_BYTE_to_DWORD(DEST[255:128], SRC[63:32])
DEST[MAXVL-1:256] := 0
```

VPMOVSXBQ (VEX.256 Encoded Version)

```
Packed_Sign_Extend_BYTE_to_QWORD(DEST[127:0], SRC[15:0])
Packed_Sign_Extend_BYTE_to_QWORD(DEST[255:128], SRC[31:16])
DEST[MAXVL-1:256] := 0
```

VPMOVSXWD (VEX.256 Encoded Version)

```
Packed_Sign_Extend_WORD_to_DWORD(DEST[127:0], SRC[63:0])
Packed_Sign_Extend_WORD_to_DWORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] := 0
```

VPMOVSXWQ (VEX.256 Encoded Version)

```
Packed_Sign_Extend_WORD_to_QWORD(DEST[127:0], SRC[31:0])
Packed_Sign_Extend_WORD_to_QWORD(DEST[255:128], SRC[63:32])
DEST[MAXVL-1:256] := 0
```

VPMOVSXDQ (VEX.256 Encoded Version)

```
Packed_Sign_Extend_DWORD_to_QWORD(DEST[127:0], SRC[63:0])
Packed_Sign_Extend_DWORD_to_QWORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] := 0
```

VPMOVSXBW (VEX.128 Encoded Version)
Packed_Sign_Extend_BYTE_to_WORD(DEST[127:0], SRC[63:0])
DEST[MAXVL-1:128] := 0

VPMOVSXBD (VEX.128 Encoded Version)
Packed_Sign_Extend_BYTE_to_DWORD(DEST[127:0], SRC[31:0])
DEST[MAXVL-1:128] := 0

VPMOVSXBQ (VEX.128 Encoded Version)
Packed_Sign_Extend_BYTE_to_QWORD(DEST[127:0], SRC[15:0])
DEST[MAXVL-1:128] := 0

VPMOVSXWD (VEX.128 Encoded Version)
Packed_Sign_Extend_WORD_to_DWORD(DEST[127:0], SRC[63:0])
DEST[MAXVL-1:128] := 0

VPMOVSXWQ (VEX.128 Encoded Version)
Packed_Sign_Extend_WORD_to_QWORD(DEST[127:0], SRC[31:0])
DEST[MAXVL-1:128] := 0

VPMOVSXDQ (VEX.128 Encoded Version)
Packed_Sign_Extend_DWORD_to_QWORD(DEST[127:0], SRC[63:0])
DEST[MAXVL-1:128] := 0

PMOVSXBW
Packed_Sign_Extend_BYTE_to_WORD(DEST[127:0], SRC[63:0])
DEST[MAXVL-1:128] (Unmodified)

PMOVSXBD
Packed_Sign_Extend_BYTE_to_DWORD(DEST[127:0], SRC[31:0])
DEST[MAXVL-1:128] (Unmodified)

PMOVSXBQ
Packed_Sign_Extend_BYTE_to_QWORD(DEST[127:0], SRC[15:0])
DEST[MAXVL-1:128] (Unmodified)

PMOVSXWD
Packed_Sign_Extend_WORD_to_DWORD(DEST[127:0], SRC[63:0])
DEST[MAXVL-1:128] (Unmodified)

PMOVSXWQ
Packed_Sign_Extend_WORD_to_QWORD(DEST[127:0], SRC[31:0])
DEST[MAXVL-1:128] (Unmodified)

PMOVSXDQ
Packed_Sign_Extend_DWORD_to_QWORD(DEST[127:0], SRC[63:0])
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VPMOVSXBW __m512i _mm512_cvtepi8_epi16(__m256i a);
VPMOVSXBW __m512i _mm512_mask_cvtepi8_epi16(__m512i src, __mmask32 k, __m256i a);
VPMOVSXBW __m512i _mm512_maskz_cvtepi8_epi16(__mmask32 k, __m256i a);
VPMOVSXBD __m512i _mm512_cvtepi8_epi32(__m128i a);

VPMOVSXBD __m512i _mm512_mask_cvtepi8_epi32(__m512i src, __mmask16 k, __m128i a);
VPMOVSXBD __m512i _mm512_maskz_cvtepi8_epi32(__mmask16 k, __m128i a);
VPMOVSXBQ __m512i _mm512_cvtepi8_epi64(__m128i a);
VPMOVSXBQ __m512i _mm512_mask_cvtepi8_epi64(__m512i src, __mmask8 k, __m128i a);
VPMOVSXBQ __m512i _mm512_maskz_cvtepi8_epi64(__mmask8 k, __m128i a);
VPMOVSXDQ __m512i _mm512_cvtepi32_epi64(__m256i a);
VPMOVSXDQ __m512i _mm512_mask_cvtepi32_epi64(__m512i src, __mmask8 k, __m256i a);
VPMOVSXDQ __m512i _mm512_maskz_cvtepi32_epi64(__mmask8 k, __m256i a);
VPMOVSXWD __m512i _mm512_cvtepi16_epi32(__m256i a);
VPMOVSXWD __m512i _mm512_mask_cvtepi16_epi32(__m512i src, __mmask16 k, __m256i a);
VPMOVSXWD __m512i _mm512_maskz_cvtepi16_epi32(__mmask16 k, __m256i a);
VPMOVSXWQ __m512i _mm512_cvtepi16_epi64(__m128i a);
VPMOVSXWQ __m512i _mm512_mask_cvtepi16_epi64(__m512i src, __mmask8 k, __m128i a);
VPMOVSXWQ __m512i _mm512_maskz_cvtepi16_epi64(__mmask8 k, __m128i a);
VPMOVSXBW __m256i _mm256_cvtepi8_epi16(__m128i a);
VPMOVSXBW __m256i _mm256_mask_cvtepi8_epi16(__m256i src, __mmask16 k, __m128i a);
VPMOVSXBW __m256i _mm256_maskz_cvtepi8_epi16(__mmask16 k, __m128i a);
VPMOVSXBD __m256i _mm256_cvtepi8_epi32(__m128i a);
VPMOVSXBD __m256i _mm256_mask_cvtepi8_epi32(__m256i src, __mmask8 k, __m128i a);
VPMOVSXBD __m256i _mm256_maskz_cvtepi8_epi32(__mmask8 k, __m128i a);
VPMOVSXBQ __m256i _mm256_cvtepi8_epi64(__m128i a);
VPMOVSXBQ __m256i _mm256_mask_cvtepi8_epi64(__m256i src, __mmask8 k, __m128i a);
VPMOVSXBQ __m256i _mm256_maskz_cvtepi8_epi64(__mmask8 k, __m128i a);
VPMOVSXDQ __m256i _mm256_cvtepi32_epi64(__m128i a);
VPMOVSXDQ __m256i _mm256_mask_cvtepi32_epi64(__m256i src, __mmask8 k, __m128i a);
VPMOVSXDQ __m256i _mm256_maskz_cvtepi32_epi64(__mmask8 k, __m128i a);
VPMOVSXWD __m256i _mm256_cvtepi16_epi32(__m128i a);
VPMOVSXWD __m256i _mm256_mask_cvtepi16_epi32(__m256i src, __mmask8 k, __m128i a);
VPMOVSXWD __m256i _mm256_maskz_cvtepi16_epi32(__mmask8 k, __m128i a);
VPMOVSXWQ __m256i _mm256_cvtepi16_epi64(__m128i a);
VPMOVSXWQ __m256i _mm256_mask_cvtepi16_epi64(__m256i src, __mmask8 k, __m128i a);
VPMOVSXWQ __m256i _mm256_maskz_cvtepi16_epi64(__mmask8 k, __m128i a);
VPMOVSXBW __m128i _mm_mask_cvtepi8_epi16(__m128i src, __mmask8 k, __m128i a);
VPMOVSXBW __m128i _mm_maskz_cvtepi8_epi16(__mmask8 k, __m128i a);
VPMOVSXBD __m128i _mm_mask_cvtepi8_epi32(__m128i src, __mmask8 k, __m128i a);
VPMOVSXBD __m128i _mm_maskz_cvtepi8_epi32(__mmask8 k, __m128i a);
VPMOVSXBQ __m128i _mm_mask_cvtepi8_epi64(__m128i src, __mmask8 k, __m128i a);
VPMOVSXBQ __m128i _mm_maskz_cvtepi8_epi64(__mmask8 k, __m128i a);
VPMOVSXDQ __m128i _mm_mask_cvtepi32_epi64(__m128i src, __mmask8 k, __m128i a);
VPMOVSXDQ __m128i _mm_maskz_cvtepi32_epi64(__mmask8 k, __m128i a);
VPMOVSXWD __m128i _mm_mask_cvtepi16_epi32(__m128i src, __mmask8 k, __m128i a);
VPMOVSXWD __m128i _mm_maskz_cvtepi16_epi32(__mmask8 k, __m128i a);
VPMOVSXWQ __m128i _mm_mask_cvtepi16_epi64(__m128i src, __mmask8 k, __m128i a);
VPMOVSXWQ __m128i _mm_maskz_cvtepi16_epi64(__mmask8 k, __m128i a);
PMOVSXBW __m128i _mm_ cvtepi8_epi16 (__m128i a);
PMOVSXBD __m128i _mm_ cvtepi8_epi32 (__m128i a);
PMOVSXBQ __m128i _mm_ cvtepi8_epi64 (__m128i a);
PMOVSXWD __m128i _mm_ cvtepi16_epi32 (__m128i a);
PMOVSXWQ __m128i _mm_ cvtepi16_epi64 (__m128i a);
PMOVSXDQ __m128i _mm_ cvtepi32_epi64 (__m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-53, “Type E5 Class Exception Conditions.”
Additionally:

#UD	If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.

PMOVZX—Packed Move With Zero Extend

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0f 38 30 /r PMOVZXBW xmm1, xmm2/m64	A	V/V	SSE4_1	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
66 0f 38 31 /r PMOVZXBD xmm1, xmm2/m32	A	V/V	SSE4_1	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
66 0f 38 32 /r PMOVZXBQ xmm1, xmm2/m16	A	V/V	SSE4_1	Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
66 0f 38 33 /r PMOVZXWD xmm1, xmm2/m64	A	V/V	SSE4_1	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
66 0f 38 34 /r PMOVZXWQ xmm1, xmm2/m32	A	V/V	SSE4_1	Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
66 0f 38 35 /r PMOVZXDQ xmm1, xmm2/m64	A	V/V	SSE4_1	Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 30 /r VPMOVZXBW xmm1, xmm2/m64	A	V/V	AVX	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
VEX.128.66.0F38.WIG 31 /r VPMOVZXBD xmm1, xmm2/m32	A	V/V	AVX	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 32 /r VPMOVZXBQ xmm1, xmm2/m16	A	V/V	AVX	Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 33 /r VPMOVZXWD xmm1, xmm2/m64	A	V/V	AVX	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 34 /r VPMOVZXWQ xmm1, xmm2/m32	A	V/V	AVX	Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F 38.WIG 35 /r VPMOVZXDQ xmm1, xmm2/m64	A	V/V	AVX	Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.256.66.0F38.WIG 30 /r VPMOVZXBW ymm1, xmm2/m128	A	V/V	AVX2	Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.
VEX.256.66.0F38.WIG 31 /r VPMOVZXBD ymm1, xmm2/m64	A	V/V	AVX2	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1.
VEX.256.66.0F38.WIG 32 /r VPMOVZXBQ ymm1, xmm2/m32	A	V/V	AVX2	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1.
VEX.256.66.0F38.WIG 33 /r VPMOVZXWD ymm1, xmm2/m128	A	V/V	AVX2	Zero extend 8 packed 16-bit integers xmm2/m128 to 8 packed 32-bit integers in ymm1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	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 ymm1.
VEX.256.66.0F38.WIG 35 /r VPMOVZXDQ ymm1, xmm2/m128	A	V/V	AVX2	Zero extend 4 packed 32-bit integers in xmm2/m128 to 4 packed 64-bit integers in ymm1.
EVEX.128.66.0F38 30.WIG /r VPMOVZXBW xmm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 30 /r VPMOVZXBW ymm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 30 /r VPMOVZXBW zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512BW OR AVX10.1	Zero extend 32 packed 8-bit integers in ymm2/m256 to 32 packed 16-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 31 /r VPMOVZXBD xmm1 {k1}{z}, xmm2/m32	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 31 /r VPMOVZXBD ymm1 {k1}{z}, xmm2/m64	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 31 /r VPMOVZXBD zmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512F OR AVX10.1	Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 32 /r VPMOVZXBQ xmm1 {k1}{z}, xmm2/m16	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 32 /r VPMOVZXBQ ymm1 {k1}{z}, xmm2/m32	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 32 /r VPMOVZXBQ zmm1 {k1}{z}, xmm2/m64	D	V/V	AVX512F OR AVX10.1	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 33 /r VPMOVZXWD xmm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 33 /r VPMOVZXWD ymm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Zero extend 8 packed 16-bit integers in xmm2/m128 to 8 packed 32-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 33 /r VPMOVZXWD zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512F OR AVX10.1	Zero extend 16 packed 16-bit integers in ymm2/m256 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 34 /r VPMOVZXWQ xmm1 {k1}{z}, xmm2/m32	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 34 /r VPMOVZXWQ ymm1 {k1}{z}, xmm2/m64	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.WIG 34 /r VPMOVZXWQ zmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512F OR AVX10.1	Zero extend 8 packed 16-bit integers in xmm2/m128 to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.W0 35 /r VPMOVZXDQ xmm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 35 /r VPMOVZXDQ ymm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Zero extend 4 packed 32-bit integers in xmm2/m128 to 4 packed 64-bit integers in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 35 /r VPMOVZXDQ zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512F OR AVX10.1	Zero extend 8 packed 32-bit integers in ymm2/m256 to 8 packed 64-bit integers in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Half Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
C	Quarter Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
D	Eighth Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Legacy and VEX encoded versions: Packed byte, word, or dword integers starting from the low bytes of the source operand (second operand) are zero extended to packed word, dword, or quadword integers in the destination operand.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: Packed byte, word, or dword integers starting from the low bytes of the source operand (second operand) are zero extended to packed word, dword, or quadword integers in the destination operand under the writemask.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

Packed_Zero_Extend_BYTE_to_WORD(DEST, SRC)

```
DEST[15:0] := ZeroExtend(SRC[7:0]);
DEST[31:16] := ZeroExtend(SRC[15:8]);
DEST[47:32] := ZeroExtend(SRC[23:16]);
DEST[63:48] := ZeroExtend(SRC[31:24]);
DEST[79:64] := ZeroExtend(SRC[39:32]);
DEST[95:80] := ZeroExtend(SRC[47:40]);
DEST[111:96] := ZeroExtend(SRC[55:48]);
DEST[127:112] := ZeroExtend(SRC[63:56]);
```

Packed_Zero_Extend_BYTE_to_DWORD(DEST, SRC)

```
DEST[31:0] := ZeroExtend(SRC[7:0]);
DEST[63:32] := ZeroExtend(SRC[15:8]);
DEST[95:64] := ZeroExtend(SRC[23:16]);
DEST[127:96] := ZeroExtend(SRC[31:24]);
```

Packed_Zero_Extend_BYTE_to_QWORD(DEST, SRC)

```
DEST[63:0] := ZeroExtend(SRC[7:0]);
DEST[127:64] := ZeroExtend(SRC[15:8]);
```

Packed_Zero_Extend_WORD_to_DWORD(DEST, SRC)

```
DEST[31:0] := ZeroExtend(SRC[15:0]);
DEST[63:32] := ZeroExtend(SRC[31:16]);
DEST[95:64] := ZeroExtend(SRC[47:32]);
DEST[127:96] := ZeroExtend(SRC[63:48]);
```

Packed_Zero_Extend_WORD_to_QWORD(DEST, SRC)

```
DEST[63:0] := ZeroExtend(SRC[15:0]);
DEST[127:64] := ZeroExtend(SRC[31:16]);
```

Packed_Zero_Extend_DWORD_to_QWORD(DEST, SRC)

```
DEST[63:0] := ZeroExtend(SRC[31:0]);
DEST[127:64] := ZeroExtend(SRC[63:32]);
```

VPMOVZXBW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
Packed_Zero_Extend_BYTE_to_WORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Zero_Extend_BYTE_to_WORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Zero_Extend_BYTE_to_WORD(TMP_DEST[383:256], SRC[191:128])
    Packed_Zero_Extend_BYTE_to_WORD(TMP_DEST[511:384], SRC[255:192])
FI;
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TEMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVZXBD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
    Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
    Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[383:256], SRC[95:64])
    Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TEMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVZXBQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[127:0], SRC[15:0])
IF VL >= 256
    Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[255:128], SRC[31:16])
FI;
IF VL >= 512
    Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[383:256], SRC[47:32])
    Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[511:384], SRC[63:48])
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVZXWD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[383:256], SRC[191:128])
    Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[511:384], SRC[256:192])
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TEMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVZXWQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
    Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
    Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[383:256], SRC[95:64])
    Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVZXDQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[383:256], SRC[191:128])
    Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[511:384], SRC[255:192])
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVZXBW (VEX.256 Encoded Version)

```
Packed_Zero_Extend_BYTE_to_WORD(DEST[127:0], SRC[63:0])
Packed_Zero_Extend_BYTE_to_WORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] := 0
```

VPMOVZXBD (VEX.256 Encoded Version)

```
Packed_Zero_Extend_BYTE_to_DWORD(DEST[127:0], SRC[31:0])
Packed_Zero_Extend_BYTE_to_DWORD(DEST[255:128], SRC[63:32])
DEST[MAXVL-1:256] := 0
```

VPMOVZXBQ (VEX.256 Encoded Version)

```
Packed_Zero_Extend_BYTE_to_QWORD(DEST[127:0], SRC[15:0])
Packed_Zero_Extend_BYTE_to_QWORD(DEST[255:128], SRC[31:16])
DEST[MAXVL-1:256] := 0
```

VPMOVZXWD (VEX.256 Encoded Version)

```
Packed_Zero_Extend_WORD_to_DWORD(DEST[127:0], SRC[63:0])
Packed_Zero_Extend_WORD_to_DWORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] := 0
```

VPMOVZXWQ (VEX.256 Encoded Version)

```
Packed_Zero_Extend_WORD_to_QWORD(DEST[127:0], SRC[31:0])
Packed_Zero_Extend_WORD_to_QWORD(DEST[255:128], SRC[63:32])
DEST[MAXVL-1:256] := 0
```

VPMOVZXDQ (VEX.256 Encoded Version)

```
Packed_Zero_Extend_DWORD_to_QWORD(DEST[127:0], SRC[63:0])
Packed_Zero_Extend_DWORD_to_QWORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] := 0
```

VPMOVZXBW (VEX.128 Encoded Version)
Packed_Zero_Extend_BYTE_to_WORD(DEST[127:0], SRC[63:0])
DEST[MAXVL-1:128] := 0

VPMOVZXBD (VEX.128 Encoded Version)
Packed_Zero_Extend_BYTE_to_DWORD(DEST[127:0], SRC[31:0])
DEST[MAXVL-1:128] := 0

VPMOVZXBQ (VEX.128 Encoded Version)
Packed_Zero_Extend_BYTE_to_QWORD(DEST[127:0], SRC[15:0])
DEST[MAXVL-1:128] := 0

VPMOVZXWD (VEX.128 Encoded Version)
Packed_Zero_Extend_WORD_to_DWORD(DEST[127:0], SRC[63:0])
DEST[MAXVL-1:128] := 0

VPMOVZXWQ (VEX.128 Encoded Version)
Packed_Zero_Extend_WORD_to_QWORD(DEST[127:0], SRC[31:0])
DEST[MAXVL-1:128] := 0

VPMOVZXDQ (VEX.128 Encoded Version)
Packed_Zero_Extend_DWORD_to_QWORD(DEST[127:0], SRC[63:0])
DEST[MAXVL-1:128] := 0

PMOVZXBW
Packed_Zero_Extend_BYTE_to_WORD(DEST[127:0], SRC[63:0])
DEST[MAXVL-1:128] (Unmodified)

PMOVZXBD
Packed_Zero_Extend_BYTE_to_DWORD(DEST[127:0], SRC[31:0])
DEST[MAXVL-1:128] (Unmodified)

PMOVZXBQ
Packed_Zero_Extend_BYTE_to_QWORD(DEST[127:0], SRC[15:0])
DEST[MAXVL-1:128] (Unmodified)

PMOVZXWD
Packed_Zero_Extend_WORD_to_DWORD(DEST[127:0], SRC[63:0])
DEST[MAXVL-1:128] (Unmodified)

PMOVZXWQ
Packed_Zero_Extend_WORD_to_QWORD(DEST[127:0], SRC[31:0])
DEST[MAXVL-1:128] (Unmodified)

PMOVZXDQ
Packed_Zero_Extend_DWORD_to_QWORD(DEST[127:0], SRC[63:0])
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VPMOVZXBW __m512i _mm512_cvtepu8_epi16(__m256i a);
VPMOVZXBW __m512i _mm512_mask_cvtepu8_epi16(__m512i src, __mmask32 k, __m256i a);
VPMOVZXBW __m512i _mm512_maskz_cvtepu8_epi16(__mmask32 k, __m256i a);
VPMOVZXBD __m512i _mm512_cvtepu8_epi32(__m128i a);

VPMOVZXBD __m512i _mm512_mask_cvtepu8_epi32(__m512i src, __mmask16 k, __m128i a);
VPMOVZXBD __m512i _mm512_maskz_cvtepu8_epi32(__mmask16 k, __m128i a);
VPMOVZXBQ __m512i _mm512_cvtepu8_epi64(__m128i a);
VPMOVZXBQ __m512i _mm512_mask_cvtepu8_epi64(__m512i src, __mmask8 k, __m128i a);
VPMOVZXBQ __m512i _mm512_maskz_cvtepu8_epi64(__mmask8 k, __m128i a);
VPMOVZXDQ __m512i _mm512_cvtepu32_epi64(__m256i a);
VPMOVZXDQ __m512i _mm512_mask_cvtepu32_epi64(__m512i src, __mmask8 k, __m256i a);
VPMOVZXDQ __m512i _mm512_maskz_cvtepu32_epi64(__mmask8 k, __m256i a);
VPMOVZXWD __m512i _mm512_cvtepu16_epi32(__m256i a);
VPMOVZXWD __m512i _mm512_mask_cvtepu16_epi32(__m512i src, __mmask16 k, __m256i a);
VPMOVZXWD __m512i _mm512_maskz_cvtepu16_epi32(__mmask16 k, __m256i a);
VPMOVZXWQ __m512i _mm512_cvtepu16_epi64(__m128i a);
VPMOVZXWQ __m512i _mm512_mask_cvtepu16_epi64(__m512i src, __mmask8 k, __m128i a);
VPMOVZXWQ __m512i _mm512_maskz_cvtepu16_epi64(__mmask8 k, __m128i a);
VPMOVZXBW __m256i _mm256_cvtepu8_epi16(__m128i a);
VPMOVZXBW __m256i _mm256_mask_cvtepu8_epi16(__m256i src, __mmask16 k, __m128i a);
VPMOVZXBW __m256i _mm256_maskz_cvtepu8_epi16(__mmask16 k, __m128i a);
VPMOVZXBD __m256i _mm256_cvtepu8_epi32(__m128i a);
VPMOVZXBD __m256i _mm256_mask_cvtepu8_epi32(__m256i src, __mmask8 k, __m128i a);
VPMOVZXBD __m256i _mm256_maskz_cvtepu8_epi32(__mmask8 k, __m128i a);
VPMOVZXBQ __m256i _mm256_cvtepu8_epi64(__m128i a);
VPMOVZXBQ __m256i _mm256_mask_cvtepu8_epi64(__m256i src, __mmask8 k, __m128i a);
VPMOVZXBQ __m256i _mm256_maskz_cvtepu8_epi64(__mmask8 k, __m128i a);
VPMOVZXDQ __m256i _mm256_cvtepu32_epi64(__m128i a);
VPMOVZXDQ __m256i _mm256_mask_cvtepu32_epi64(__m256i src, __mmask8 k, __m128i a);
VPMOVZXDQ __m256i _mm256_maskz_cvtepu32_epi64(__mmask8 k, __m128i a);
VPMOVZXWD __m256i _mm256_cvtepu16_epi32(__m128i a);
VPMOVZXWD __m256i _mm256_mask_cvtepu16_epi32(__m256i src, __mmask8 k, __m128i a);
VPMOVZXWD __m256i _mm256_maskz_cvtepu16_epi32(__mmask8 k, __m128i a);
VPMOVZXWQ __m256i _mm256_cvtepu16_epi64(__m128i a);
VPMOVZXWQ __m256i _mm256_mask_cvtepu16_epi64(__m256i src, __mmask8 k, __m128i a);
VPMOVZXWQ __m256i _mm256_maskz_cvtepu16_epi64(__mmask8 k, __m128i a);
VPMOVZXBW __m128i _mm_mask_cvtepu8_epi16(__m128i src, __mmask8 k, __m128i a);
VPMOVZXBW __m128i _mm_maskz_cvtepu8_epi16(__mmask8 k, __m128i a);
VPMOVZXBD __m128i _mm_mask_cvtepu8_epi32(__m128i src, __mmask8 k, __m128i a);
VPMOVZXBD __m128i _mm_maskz_cvtepu8_epi32(__mmask8 k, __m128i a);
VPMOVZXBQ __m128i _mm_mask_cvtepu8_epi64(__m128i src, __mmask8 k, __m128i a);
VPMOVZXBQ __m128i _mm_maskz_cvtepu8_epi64(__mmask8 k, __m128i a);
VPMOVZXDQ __m128i _mm_mask_cvtepu32_epi64(__m128i src, __mmask8 k, __m128i a);
VPMOVZXDQ __m128i _mm_maskz_cvtepu32_epi64(__mmask8 k, __m128i a);
VPMOVZXWD __m128i _mm_mask_cvtepu16_epi32(__m128i src, __mmask8 k, __m128i a);
VPMOVZXWD __m128i _mm_maskz_cvtepu16_epi32(__mmask8 k, __m128i a);
VPMOVZXWQ __m128i _mm_mask_cvtepu16_epi64(__m128i src, __mmask8 k, __m128i a);
VPMOVZXWQ __m128i _mm_maskz_cvtepu16_epi64(__mmask8 k, __m128i a);
PMOVZXBW __m128i _mm_ cvtepu8_epi16 (__m128i a);
PMOVZXBD __m128i _mm_ cvtepu8_epi32 (__m128i a);
PMOVZXBQ __m128i _mm_ cvtepu8_epi64 (__m128i a);
PMOVZXWD __m128i _mm_ cvtepu16_epi32 (__m128i a);
PMOVZXWQ __m128i _mm_ cvtepu16_epi64 (__m128i a);
PMOVZXDQ __m128i _mm_ cvtepu32_epi64 (__m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-53, “Type E5 Class Exception Conditions.”
Additionally:

#UD	If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.

PMULDQ—Multiply Packed Doubleword Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 28 /r PMULDQ xmm1, xmm2/m128	A	V/V	SSE4_1	Multiply packed signed doubleword integers in xmm1 by packed signed doubleword integers in xmm2/m128, and store the quadword results in xmm1.
VEX.128.66.0F38.WIG 28 /r VPMULDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply packed signed doubleword integers in xmm2 by packed signed doubleword integers in xmm3/m128, and store the quadword results in xmm1.
VEX.256.66.0F38.WIG 28 /r VPMULDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply packed signed doubleword integers in ymm2 by packed signed doubleword integers in ymm3/m256, and store the quadword results in ymm1.
EVEX.128.66.0F38.W1 28 /r VPMULDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed signed doubleword integers in xmm2 by packed signed doubleword integers in xmm3/m128/m64bcst, and store the quadword results in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 28 /r VPMULDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed signed doubleword integers in ymm2 by packed signed doubleword integers in ymm3/m256/m64bcst, and store the quadword results in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 28 /r VPMULDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Multiply packed signed doubleword integers in zmm2 by packed signed doubleword integers in zmm3/m512/m64bcst, and store the quadword results in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies packed signed doubleword integers in the even-numbered (zero-based reference) elements of the first source operand with the packed signed doubleword integers in the corresponding elements of the second source operand and stores packed signed quadword results in the destination operand.

128-bit Legacy SSE version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e., the first (low) and third doubleword element. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand and the destination XMM operand is the same. The second source operand can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e., the first (low) and third doubleword element. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation.The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e., the first, 3rd, 5th, 7th doubleword element. For 256-bit memory operands, 256 bits are fetched from memory, but only the four even-numbered doublewords are used in the computation. The first source operand and the destination operand are YMM registers. The second source operand can be a YMM register or 256-bit memory location. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

EVEX encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands. The first source operand is a ZMM/YMM/XMM registers. The second source operand can be an ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination is a ZMM/YMM/XMM register, and updated according to the writemask at 64-bit granularity.

Operation

VPMULDQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := SignExtend64( SRC1[i+31:i]) * SignExtend64( SRC2[31:0])
                ELSE DEST[i+63:i] := SignExtend64( SRC1[i+31:i]) * SignExtend64( SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMULDQ (VEX.256 Encoded Version)

```
DEST[63:0] := SignExtend64( SRC1[31:0]) * SignExtend64( SRC2[31:0])
DEST[127:64] := SignExtend64( SRC1[95:64]) * SignExtend64( SRC2[95:64])
DEST[191:128] := SignExtend64( SRC1[159:128]) * SignExtend64( SRC2[159:128])
DEST[255:192] := SignExtend64( SRC1[223:192]) * SignExtend64( SRC2[223:192])
DEST[MAXVL-1:256] := 0
```

VPMULDQ (VEX.128 Encoded Version)

```
DEST[63:0] := SignExtend64( SRC1[31:0]) * SignExtend64( SRC2[31:0])
DEST[127:64] := SignExtend64( SRC1[95:64]) * SignExtend64( SRC2[95:64])
DEST[MAXVL-1:128] := 0
```

PMULDQ (128-bit Legacy SSE Version)

```
DEST[63:0] := SignExtend64( DEST[31:0]) * SignExtend64( SRC[31:0])
DEST[127:64] := SignExtend64( DEST[95:64]) * SignExtend64( SRC[95:64])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMULDQ __m512i _mm512_mul_epi32(__m512i a, __m512i b);
VPMULDQ __m512i _mm512_mask_mul_epi32(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMULDQ __m512i _mm512_maskz_mul_epi32(__mmask8 k, __m512i a, __m512i b);
VPMULDQ __m256i _mm256_mask_mul_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULDQ __m256i _mm256_mask_mul_epi32(__mmask8 k, __m256i a, __m256i b);
VPMULDQ __m128i _mm_mask_mul_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULDQ __m128i _mm_mask_mul_epi32(__mmask8 k, __m128i a, __m128i b);
(V)PMULDQ __m128i _mm_mul_epi32(__m128i a, __m128i b);
VPMULDQ __m256i _mm256_mul_epi32(__m256i a, __m256i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-51, "Type E4 Class Exception Conditions."

PMULHRSW—Packed Multiply High With Round and Scale

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 0B /r[1] PMULHRSW mm1, mm2/m64	A	V/V	SSSE3	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to mm1.
66 0F 38 0B /r PMULHRSW xmm1, xmm2/m128	A	V/V	SSSE3	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to xmm1.
VEX.128.66.0F38.WIG 0B /r VPMULHRSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to xmm1.
VEX.256.66.0F38.WIG 0B /r VPMULHRSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to ymm1.
EVEX.128.66.0F38.WIG 0B /r VPMULHRSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to xmm1 under writemask k1.
EVEX.256.66.0F38.WIG 0B /r VPMULHRSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to ymm1 under writemask k1.
EVEX.512.66.0F38.WIG 0B /r VPMULHRSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

PMULHRSW multiplies vertically each signed 16-bit integer from the destination operand (first operand) with the corresponding signed 16-bit integer of the source operand (second operand), producing intermediate, signed 32-bit integers. Each intermediate 32-bit integer is truncated to the 18 most significant bits. Rounding is always performed by adding 1 to the least significant bit of the 18-bit intermediate result. The final result is obtained by selecting the 16 bits immediately to the right of the most significant bit of each 18-bit intermediate result and packed to the destination operand.

When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15 registers.

Legacy SSE version 64-bit operand: Both operands can be MMX registers. The second source operand is an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Operation

PMULHRSW (With 64-bit Operands)

```
temp0[31:0] = INT32 ((DEST[15:0] * SRC[15:0]) >>14) + 1;
temp1[31:0] = INT32 ((DEST[31:16] * SRC[31:16]) >>14) + 1;
temp2[31:0] = INT32 ((DEST[47:32] * SRC[47:32]) >> 14) + 1;
temp3[31:0] = INT32 ((DEST[63:48] * SRc[63:48]) >> 14) + 1;
DEST[15:0] = temp0[16:1];
DEST[31:16] = temp1[16:1];
DEST[47:32] = temp2[16:1];
DEST[63:48] = temp3[16:1];
```

PMULHRSW (With 128-bit Operands)

```
temp0[31:0] = INT32 ((DEST[15:0] * SRC[15:0]) >>14) + 1;
temp1[31:0] = INT32 ((DEST[31:16] * SRC[31:16]) >>14) + 1;
temp2[31:0] = INT32 ((DEST[47:32] * SRC[47:32]) >>14) + 1;
temp3[31:0] = INT32 ((DEST[63:48] * SRC[63:48]) >>14) + 1;
temp4[31:0] = INT32 ((DEST[79:64] * SRC[79:64]) >>14) + 1;
temp5[31:0] = INT32 ((DEST[95:80] * SRC[95:80]) >>14) + 1;
temp6[31:0] = INT32 ((DEST[111:96] * SRC[111:96]) >>14) + 1;
temp7[31:0] = INT32 ((DEST[127:112] * SRC[127:112) >>14) + 1;
DEST[15:0] = temp0[16:1];
DEST[31:16] = temp1[16:1];
DEST[47:32] = temp2[16:1];
DEST[63:48] = temp3[16:1];
DEST[79:64] = temp4[16:1];
DEST[95:80] = temp5[16:1];
DEST[111:96] = temp6[16:1];
DEST[127:112] = temp7[16:1];
```

VPMULHRSW (VEX.128 Encoded Version)

```
temp0[31:0] := INT32 ((SRC1[15:0] * SRC2[15:0]) >>14) + 1
temp1[31:0] := INT32 ((SRC1[31:16] * SRC2[31:16]) >>14) + 1
temp2[31:0] := INT32 ((SRC1[47:32] * SRC2[47:32]) >>14) + 1
temp3[31:0] := INT32 ((SRC1[63:48] * SRC2[63:48]) >>14) + 1
temp4[31:0] := INT32 ((SRC1[79:64] * SRC2[79:64]) >>14) + 1
temp5[31:0] := INT32 ((SRC1[95:80] * SRC2[95:80]) >>14) + 1
temp6[31:0] := INT32 ((SRC1[111:96] * SRC2[111:96]) >>14) + 1
temp7[31:0] := INT32 ((SRC1[127:112] * SRC2[127:112) >>14) + 1
DEST[15:0] := temp0[16:1]
DEST[31:16] := temp1[16:1]
DEST[47:32] := temp2[16:1]
```

```
DEST[63:48] := temp3[16:1]
DEST[79:64] := temp4[16:1]
DEST[95:80] := temp5[16:1]
DEST[111:96] := temp6[16:1]
DEST[127:112] := temp7[16:1]
DEST[MAXVL-1:128] := 0
```

VPMULHRSW (VEX.256 Encoded Version)

```
temp0[31:0] := INT32 ((SRC1[15:0] * SRC2[15:0]) >>14) + 1
temp1[31:0] := INT32 ((SRC1[31:16] * SRC2[31:16]) >>14) + 1
temp2[31:0] := INT32 ((SRC1[47:32] * SRC2[47:32]) >>14) + 1
temp3[31:0] := INT32 ((SRC1[63:48] * SRC2[63:48]) >>14) + 1
temp4[31:0] := INT32 ((SRC1[79:64] * SRC2[79:64]) >>14) + 1
temp5[31:0] := INT32 ((SRC1[95:80] * SRC2[95:80]) >>14) + 1
temp6[31:0] := INT32 ((SRC1[111:96] * SRC2[111:96]) >>14) + 1
temp7[31:0] := INT32 ((SRC1[127:112] * SRC2[127:112) >>14) + 1
temp8[31:0] := INT32 ((SRC1[143:128] * SRC2[143:128]) >>14) + 1
temp9[31:0] := INT32 ((SRC1[159:144] * SRC2[159:144]) >>14) + 1
temp10[31:0] := INT32 ((SRC1[75:160] * SRC2[175:160]) >>14) + 1
temp11[31:0] := INT32 ((SRC1[191:176] * SRC2[191:176]) >>14) + 1
temp12[31:0] := INT32 ((SRC1[207:192] * SRC2[207:192]) >>14) + 1
temp13[31:0] := INT32 ((SRC1[223:208] * SRC2[223:208]) >>14) + 1
temp14[31:0] := INT32 ((SRC1[239:224] * SRC2[239:224]) >>14) + 1
temp15[31:0] := INT32 ((SRC1[255:240] * SRC2[255:240) >>14) + 1

DEST[15:0] := temp0[16:1]
DEST[31:16] := temp1[16:1]
DEST[47:32] := temp2[16:1]
DEST[63:48] := temp3[16:1]
DEST[79:64] := temp4[16:1]
DEST[95:80] := temp5[16:1]
DEST[111:96] := temp6[16:1]
DEST[127:112] := temp7[16:1]
DEST[143:128] := temp8[16:1]
DEST[159:144] := temp9[16:1]
DEST[175:160] := temp10[16:1]
DEST[191:176] := temp11[16:1]
DEST[207:192] := temp12[16:1]
DEST[223:208] := temp13[16:1]
DEST[239:224] := temp14[16:1]
DEST[255:240] := temp15[16:1]
DEST[MAXVL-1:256] := 0
```

VPMULHRSW (EVEX Encoded Version)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN
            temp[31:0] := ((SRC1[i+15:i] * SRC2[i+15:i]) >>14) + 1
            DEST[i+15:i] := tmp[16:1]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMULHRSW __m512i _mm512_mulhrs_epi16(__m512i a, __m512i b);
VPMULHRSW __m512i _mm512_mask_mulhrs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHRSW __m512i _mm512_maskz_mulhrs_epi16(__mmask32 k, __m512i a, __m512i b);
VPMULHRSW __m256i _mm256_mask_mulhrs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULHRSW __m256i _mm256_maskz_mulhrs_epi16(__mmask16 k, __m256i a, __m256i b);
VPMULHRSW __m128i _mm_mask_mulhrs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULHRSW __m128i _mm_maskz_mulhrs_epi16(__mmask8 k, __m128i a, __m128i b);
PMULHRSW __m64 _mm_mulhrs_pi16 (__m64 a, __m64 b)
(V)PMULHRSW __m128i _mm_mulhrs_epi16 (__m128i a, __m128i b)
VPMULHRSW __m256i _mm256_mulhrs_epi16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

PMULHUW—Multiply Packed Unsigned Integers and Store High Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E4 /r[1] PMULHUW mm1, mm2/m64	A	V/V	SSE	Multiply the packed unsigned word integers in mm1 register and mm2/m64, and store the high 16 bits of the results in mm1.
66 0F E4 /r PMULHUW xmm1, xmm2/m128	A	V/V	SSE2	Multiply the packed unsigned word integers in xmm1 and xmm2/m128, and store the high 16 bits of the results in xmm1.
VEX.128.66.0F.WIG E4 /r VPMULHUW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed unsigned word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1.
VEX.256.66.0F.WIG E4 /r VPMULHUW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply the packed unsigned word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1.
EVEX.128.66.0F.WIG E4 /r VPMULHUW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Multiply the packed unsigned word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG E4 /r VPMULHUW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Multiply the packed unsigned word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG E4 /r VPMULHUW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Multiply the packed unsigned word integers in zmm2 and zmm3/m512, and store the high 16 bits of the results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD unsigned multiply of the packed unsigned word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each 32-bit intermediate results in the destination operand. (Figure 4-12 shows this operation when using 64-bit operands.)

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

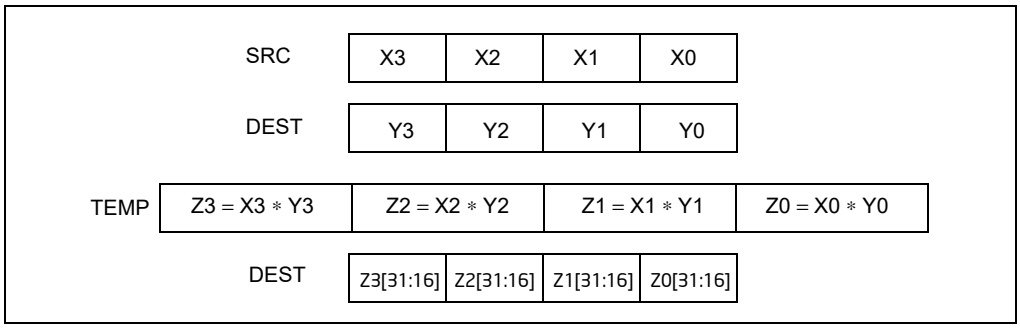


Figure 4-12. PMULHUW and PMULHW Instruction Operation Using 64-bit Operands

Operation

PMULHUW (With 64-bit Operands)

```
TEMP0[31:0] := DEST[15:0] * SRC[15:0]; (* Unsigned multiplication *)
TEMP1[31:0] := DEST[31:16] * SRC[31:16];
TEMP2[31:0] := DEST[47:32] * SRC[47:32];
TEMP3[31:0] := DEST[63:48] * SRC[63:48];
DEST[15:0] := TEMP0[31:16];
DEST[31:16] := TEMP1[31:16];
DEST[47:32] := TEMP2[31:16];
DEST[63:48] := TEMP3[31:16];
```

PMULHUW (With 128-bit Operands)

```
TEMP0[31:0] := DEST[15:0] * SRC[15:0]; (* Unsigned multiplication *)
TEMP1[31:0] := DEST[31:16] * SRC[31:16];
TEMP2[31:0] := DEST[47:32] * SRC[47:32];
TEMP3[31:0] := DEST[63:48] * SRC[63:48];
TEMP4[31:0] := DEST[79:64] * SRC[79:64];
TEMP5[31:0] := DEST[95:80] * SRC[95:80];
TEMP6[31:0] := DEST[111:96] * SRC[111:96];
TEMP7[31:0] := DEST[127:112] * SRC[127:112];
DEST[15:0] := TEMP0[31:16];
DEST[31:16] := TEMP1[31:16];
DEST[47:32] := TEMP2[31:16];
DEST[63:48] := TEMP3[31:16];
DEST[79:64] := TEMP4[31:16];
DEST[95:80] := TEMP5[31:16];
DEST[111:96] := TEMP6[31:16];
DEST[127:112] := TEMP7[31:16];
```

VPMULHUW (VEX.128 Encoded Version)

```
TEMP0[31:0] := SRC1[15:0] * SRC2[15:0]
TEMP1[31:0] := SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] := SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] := SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] := SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] := SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] := SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] := SRC1[127:112] * SRC2[127:112]
DEST[15:0] := TEMP0[31:16]
DEST[31:16] := TEMP1[31:16]
DEST[47:32] := TEMP2[31:16]
DEST[63:48] := TEMP3[31:16]
DEST[79:64] := TEMP4[31:16]
DEST[95:80] := TEMP5[31:16]
DEST[111:96] := TEMP6[31:16]
DEST[127:112] := TEMP7[31:16]
DEST[MAXVL-1:128] := 0
```

PMULHUW (VEX.256 Encoded Version)

```
TEMP0[31:0] := SRC1[15:0] * SRC2[15:0]
TEMP1[31:0] := SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] := SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] := SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] := SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] := SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] := SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] := SRC1[127:112] * SRC2[127:112]
TEMP8[31:0] := SRC1[143:128] * SRC2[143:128]
TEMP9[31:0] := SRC1[159:144] * SRC2[159:144]
TEMP10[31:0] := SRC1[175:160] * SRC2[175:160]
TEMP11[31:0] := SRC1[191:176] * SRC2[191:176]
TEMP12[31:0] := SRC1[207:192] * SRC2[207:192]
TEMP13[31:0] := SRC1[223:208] * SRC2[223:208]
TEMP14[31:0] := SRC1[239:224] * SRC2[239:224]
TEMP15[31:0] := SRC1[255:240] * SRC2[255:240]
DEST[15:0] := TEMP0[31:16]
DEST[31:16] := TEMP1[31:16]
DEST[47:32] := TEMP2[31:16]
DEST[63:48] := TEMP3[31:16]
DEST[79:64] := TEMP4[31:16]
DEST[95:80] := TEMP5[31:16]
DEST[111:96] := TEMP6[31:16]
DEST[127:112] := TEMP7[31:16]
DEST[143:128] := TEMP8[31:16]
DEST[159:144] := TEMP9[31:16]
DEST[175:160] := TEMP10[31:16]
DEST[191:176] := TEMP11[31:16]
DEST[207:192] := TEMP12[31:16]
DEST[223:208] := TEMP13[31:16]
DEST[239:224] := TEMP14[31:16]
DEST[255:240] := TEMP15[31:16]
DEST[MAXVL-1:256] := 0
```

PMULHUW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN
            temp[31:0] := SRC1[i+15:i] * SRC2[i+15:i]
            DEST[i+15:i] := tmp[31:16]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMULHUW __m512i _mm512_mulhi_epu16(__m512i a, __m512i b);
VPMULHUW __m512i _mm512_mask_mulhi_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHUW __m512i _mm512_maskz_mulhi_epu16(__mmask32 k, __m512i a, __m512i b);
VPMULHUW __m256i _mm256_mask_mulhi_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULHUW __m256i _mm256_maskz_mulhi_epu16(__mmask16 k, __m256i a, __m256i b);
VPMULHUW __m128i _mm_mask_mulhi_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULHUW __m128i _mm_maskz_mulhi_epu16(__mmask8 k, __m128i a, __m128i b);
PMULHUW __m64 _mm_mulhi_pu16(__m64 a, __m64 b)
(V)PMULHUW __m128i _mm_mulhi_epu16 (__m128i a, __m128i b)
VPMULHUW __m256i _mm256_mulhi_epu16 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, "Type E4 Class Exception Conditions."

PMULHW—Multiply Packed Signed Integers and Store High Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E5 /r[1] PMULHW mm, mm/m64	A	V/V	MMX	Multiply the packed signed word integers in mm1 register and mm2/m64, and store the high 16 bits of the results in mm1.
66 0F E5 /r PMULHW xmm1, xmm2/m128	A	V/V	SSE2	Multiply the packed signed word integers in xmm1 and xmm2/m128, and store the high 16 bits of the results in xmm1.
VEX.128.66.0F.WIG E5 /r VPMULHW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1.
VEX.256.66.0F.WIG E5 /r VPMULHW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1.
EVEX.128.66.0F.WIG E5 /r VPMULHW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG E5 /r VPMULHW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG E5 /r VPMULHW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Multiply the packed signed word integers in zmm2 and zmm3/m512, and store the high 16 bits of the results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-12 shows this operation when using 64-bit operands.)

n 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Operation

PMULHW (With 64-bit Operands)

```
TEMP0[31:0] := DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
TEMP1[31:0] := DEST[31:16] * SRC[31:16];
TEMP2[31:0] := DEST[47:32] * SRC[47:32];
TEMP3[31:0] := DEST[63:48] * SRC[63:48];
DEST[15:0] := TEMP0[31:16];
DEST[31:16] := TEMP1[31:16];
DEST[47:32] := TEMP2[31:16];
DEST[63:48] := TEMP3[31:16];
```

PMULHW (With 128-bit Operands)

```
TEMP0[31:0] := DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
TEMP1[31:0] := DEST[31:16] * SRC[31:16];
TEMP2[31:0] := DEST[47:32] * SRC[47:32];
TEMP3[31:0] := DEST[63:48] * SRC[63:48];
TEMP4[31:0] := DEST[79:64] * SRC[79:64];
TEMP5[31:0] := DEST[95:80] * SRC[95:80];
TEMP6[31:0] := DEST[111:96] * SRC[111:96];
TEMP7[31:0] := DEST[127:112] * SRC[127:112];
DEST[15:0] := TEMP0[31:16];
DEST[31:16] := TEMP1[31:16];
DEST[47:32] := TEMP2[31:16];
DEST[63:48] := TEMP3[31:16];
DEST[79:64] := TEMP4[31:16];
DEST[95:80] := TEMP5[31:16];
DEST[111:96] := TEMP6[31:16];
DEST[127:112] := TEMP7[31:16];
```

VPMULHW (VEX.128 Encoded Version)

```
TEMP0[31:0] := SRC1[15:0] * SRC2[15:0] (*Signed Multiplication*)
TEMP1[31:0] := SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] := SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] := SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] := SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] := SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] := SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] := SRC1[127:112] * SRC2[127:112]
DEST[15:0] := TEMP0[31:16]
DEST[31:16] := TEMP1[31:16]
DEST[47:32] := TEMP2[31:16]
DEST[63:48] := TEMP3[31:16]
DEST[79:64] := TEMP4[31:16]
DEST[95:80] := TEMP5[31:16]
```

DEST[111:96] := TEMP6[31:16]
DEST[127:112] := TEMP7[31:16]
DEST[MAXVL-1:128] := 0

PMULHW (VEX.256 Encoded Version)
TEMP0[31:0] := SRC1[15:0] * SRC2[15:0] (*Signed Multiplication*)
TEMP1[31:0] := SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] := SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] := SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] := SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] := SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] := SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] := SRC1[127:112] * SRC2[127:112]
TEMP8[31:0] := SRC1[143:128] * SRC2[143:128]
TEMP9[31:0] := SRC1[159:144] * SRC2[159:144]
TEMP10[31:0] := SRC1[175:160] * SRC2[175:160]
TEMP11[31:0] := SRC1[191:176] * SRC2[191:176]
TEMP12[31:0] := SRC1[207:192] * SRC2[207:192]
TEMP13[31:0] := SRC1[223:208] * SRC2[223:208]
TEMP14[31:0] := SRC1[239:224] * SRC2[239:224]
TEMP15[31:0] := SRC1[255:240] * SRC2[255:240]
DEST[15:0] := TEMP0[31:16]
DEST[31:16] := TEMP1[31:16]
DEST[47:32] := TEMP2[31:16]
DEST[63:48] := TEMP3[31:16]
DEST[79:64] := TEMP4[31:16]
DEST[95:80] := TEMP5[31:16]
DEST[111:96] := TEMP6[31:16]
DEST[127:112] := TEMP7[31:16]
DEST[143:128] := TEMP8[31:16]
DEST[159:144] := TEMP9[31:16]
DEST[175:160] := TEMP10[31:16]
DEST[191:176] := TEMP11[31:16]
DEST[207:192] := TEMP12[31:16]
DEST[223:208] := TEMP13[31:16]
DEST[239:224] := TEMP14[31:16]
DEST[255:240] := TEMP15[31:16]
DEST[MAXVL-1:256] := 0

PMULHW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN
            temp[31:0] := SRC1[i+15:i] * SRC2[i+15:i]
            DEST[i+15:i] := tmp[31:16]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMULHW __m512i _mm512_mulhi_epi16(__m512i a, __m512i b);
VPMULHW __m512i _mm512_mask_mulhi_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHW __m512i _mm512_maskz_mulhi_epi16(__mmask32 k, __m512i a, __m512i b);
VPMULHW __m256i _mm256_mask_mulhi_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULHW __m256i _mm256_maskz_mulhi_epi16(__mmask16 k, __m256i a, __m256i b);
VPMULHW __m128i _mm_mask_mulhi_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULHW __m128i _mm_maskz_mulhi_epi16(__mmask8 k, __m128i a, __m128i b);
PMULHW __m64 _mm_mulhi_pi16 (__m64 m1, __m64 m2)
(V)PMULHW __m128i _mm_mulhi_epi16 (__m128i a, __m128i b)
VPMULHW __m256i _mm256_mulhi_epi16 (__m256i a, __m256i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

PMULLD/PMULLQ—Multiply Packed Integers and Store Low Result

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 40 /r PMULLD xmm1, xmm2/m128	A	V/V	SSE4_1	Multiply the packed dword signed integers in xmm1 and xmm2/m128 and store the low 32 bits of each product in xmm1.
VEX.128.66.0F38.WIG 40 /r VPMULLD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed dword signed integers in xmm2 and xmm3/m128 and store the low 32 bits of each product in xmm1.
VEX.256.66.0F38.WIG 40 /r VPMULLD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply the packed dword signed integers in ymm2 and ymm3/m256 and store the low 32 bits of each product in ymm1.
EVEX.128.66.0F38.W0 40 /r VPMULLD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply the packed dword signed integers in xmm2 and xmm3/m128/m32bcst and store the low 32 bits of each product in xmm1 under writemask k1.
EVEX.256.66.0F38.W0 40 /r VPMULLD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply the packed dword signed integers in ymm2 and ymm3/m256/m32bcst and store the low 32 bits of each product in ymm1 under writemask k1.
EVEX.512.66.0F38.W0 40 /r VPMULLD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Multiply the packed dword signed integers in zmm2 and zmm3/m512/m32bcst and store the low 32 bits of each product in zmm1 under writemask k1.
EVEX.128.66.0F38.W1 40 /r VPMULLQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Multiply the packed qword signed integers in xmm2 and xmm3/m128/m64bcst and store the low 64 bits of each product in xmm1 under writemask k1.
EVEX.256.66.0F38.W1 40 /r VPMULLQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Multiply the packed qword signed integers in ymm2 and ymm3/m256/m64bcst and store the low 64 bits of each product in ymm1 under writemask k1.
EVEX.512.66.0F38.W1 40 /r VPMULLQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ OR AVX10.1	Multiply the packed qword signed integers in zmm2 and zmm3/m512/m64bcst and store the low 64 bits of each product in zmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD signed multiply of the packed signed dword/qword integers from each element of the first source operand with the corresponding element in the second source operand. The low 32/64 bits of each 64/128-bit intermediate results are stored to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register; The second source operand is a YMM register or 256-bit memory location. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

VPMULLQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN Temp[127:0] := SRC1[i+63:i] * SRC2[63:0]
                ELSE Temp[127:0] := SRC1[i+63:i] * SRC2[i+63:i]
            FI;
            DEST[i+63:i] := Temp[63:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMULLD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN Temp[63:0] := SRC1[i+31:i] * SRC2[31:0]
                ELSE Temp[63:0] := SRC1[i+31:i] * SRC2[i+31:i]
            FI;
            DEST[i+31:i] := Temp[31:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMULLD (VEX.256 Encoded Version)
Temp0[63:0] := SRC1[31:0] * SRC2[31:0]
Temp1[63:0] := SRC1[63:32] * SRC2[63:32]
Temp2[63:0] := SRC1[95:64] * SRC2[95:64]
Temp3[63:0] := SRC1[127:96] * SRC2[127:96]
Temp4[63:0] := SRC1[159:128] * SRC2[159:128]
Temp5[63:0] := SRC1[191:160] * SRC2[191:160]
Temp6[63:0] := SRC1[223:192] * SRC2[223:192]
Temp7[63:0] := SRC1[255:224] * SRC2[255:224]

DEST[31:0] := Temp0[31:0]
DEST[63:32] := Temp1[31:0]
DEST[95:64] := Temp2[31:0]
DEST[127:96] := Temp3[31:0]
DEST[159:128] := Temp4[31:0]
DEST[191:160] := Temp5[31:0]
DEST[223:192] := Temp6[31:0]
DEST[255:224] := Temp7[31:0]
DEST[MAXVL-1:256] := 0

VPMULLD (VEX.128 Encoded Version)
Temp0[63:0] := SRC1[31:0] * SRC2[31:0]
Temp1[63:0] := SRC1[63:32] * SRC2[63:32]
Temp2[63:0] := SRC1[95:64] * SRC2[95:64]
Temp3[63:0] := SRC1[127:96] * SRC2[127:96]
DEST[31:0] := Temp0[31:0]
DEST[63:32] := Temp1[31:0]
DEST[95:64] := Temp2[31:0]
DEST[127:96] := Temp3[31:0]
DEST[MAXVL-1:128] := 0

PMULLD (128-bit Legacy SSE Version)
Temp0[63:0] := DEST[31:0] * SRC[31:0]
Temp1[63:0] := DEST[63:32] * SRC[63:32]
Temp2[63:0] := DEST[95:64] * SRC[95:64]
Temp3[63:0] := DEST[127:96] * SRC[127:96]
DEST[31:0] := Temp0[31:0]
DEST[63:32] := Temp1[31:0]
DEST[95:64] := Temp2[31:0]
DEST[127:96] := Temp3[31:0]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VPMULLD __m512i _mm512_mullo_epi32(__m512i a, __m512i b);
VPMULLD __m512i _mm512_mask_mullo_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMULLD __m512i _mm512_maskz_mullo_epi32(__mmask16 k, __m512i a, __m512i b);
VPMULLD __m256i _mm256_mask_mullo_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULLD __m256i _mm256_maskz_mullo_epi32(__mmask8 k, __m256i a, __m256i b);
VPMULLD __m128i _mm_mask_mullo_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULLD __m128i _mm_maskz_mullo_epi32(__mmask8 k, __m128i a, __m128i b);
VPMULLD __m256i _mm256_mullo_epi32(__m256i a, __m256i b);
PMULLD __m128i _mm_mullo_epi32(__m128i a, __m128i b);
VPMULLQ __m512i _mm512_mullo_epi64(__m512i a, __m512i b);
VPMULLQ __m512i _mm512_mask_mullo_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMULLQ __m512i _mm512_maskz_mullo_epi64(__mmask8 k, __m512i a, __m512i b);
VPMULLQ __m256i _mm256_mullo_epi64(__m256i a, __m256i b);
VPMULLQ __m256i _mm256_mask_mullo_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULLQ __m256i _mm256_maskz_mullo_epi64(__mmask8 k, __m256i a, __m256i b);
VPMULLQ __m128i _mm_mullo_epi64(__m128i a, __m128i b);
VPMULLQ __m128i _mm_mask_mullo_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULLQ __m128i _mm_maskz_mullo_epi64(__mmask8 k, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

PMULLW—Multiply Packed Signed Integers and Store Low Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F D5 /r[1] PMULLW mm, mm/m64	A	V/V	MMX	Multiply the packed signed word integers in mm1 register and mm2/m64, and store the low 16 bits of the results in mm1.
66 0F D5 /r PMULLW xmm1, xmm2/m128	A	V/V	SSE2	Multiply the packed signed word integers in xmm1 and xmm2/m128, and store the low 16 bits of the results in xmm1.
VEX.128.66.0F.WIG D5 /r VPMULLW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed dword signed integers in xmm2 and xmm3/m128 and store the low 32 bits of each product in xmm1.
VEX.256.66.0F.WIG D5 /r VPMULLW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the low 16 bits of the results in ymm1.
EVEX.128.66.0F.WIG D5 /r VPMULLW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the low 16 bits of the results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG D5 /r VPMULLW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the low 16 bits of the results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG D5 /r VPMULLW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Multiply the packed signed word integers in zmm2 and zmm3/m512, and store the low 16 bits of the results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the low 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-12 shows this operation when using 64-bit operands.)

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

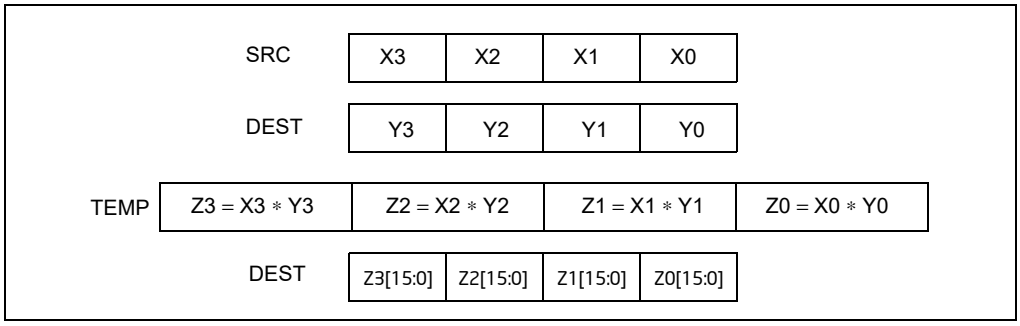


Figure 4-13. PMULLU Instruction Operation Using 64-bit Operands

Operation

PMULLW (With 64-bit Operands)

```
    TEMP0[31:0] := DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
    TEMP1[31:0] := DEST[31:16] * SRC[31:16];
    TEMP2[31:0] := DEST[47:32] * SRC[47:32];
    TEMP3[31:0] := DEST[63:48] * SRC[63:48];
    DEST[15:0] := TEMP0[15:0];
    DEST[31:16] := TEMP1[15:0];
    DEST[47:32] := TEMP2[15:0];
    DEST[63:48] := TEMP3[15:0];
```

PMULLW (With 128-bit Operands)

```
    TEMP0[31:0] := DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
    TEMP1[31:0] := DEST[31:16] * SRC[31:16];
    TEMP2[31:0] := DEST[47:32] * SRC[47:32];
    TEMP3[31:0] := DEST[63:48] * SRC[63:48];
    TEMP4[31:0] := DEST[79:64] * SRC[79:64];
    TEMP5[31:0] := DEST[95:80] * SRC[95:80];
    TEMP6[31:0] := DEST[111:96] * SRC[111:96];
    TEMP7[31:0] := DEST[127:112] * SRC[127:112];
    DEST[15:0] := TEMP0[15:0];
    DEST[31:16] := TEMP1[15:0];
    DEST[47:32] := TEMP2[15:0];
    DEST[63:48] := TEMP3[15:0];
    DEST[79:64] := TEMP4[15:0];
    DEST[95:80] := TEMP5[15:0];
    DEST[111:96] := TEMP6[15:0];
    DEST[127:112] := TEMP7[15:0];
DEST[MAXVL-1:256] := 0
```

VPMULLW (VEX.128 Encoded Version)

```
Temp0[31:0] := SRC1[15:0] * SRC2[15:0]
Temp1[31:0] := SRC1[31:16] * SRC2[31:16]
Temp2[31:0] := SRC1[47:32] * SRC2[47:32]
Temp3[31:0] := SRC1[63:48] * SRC2[63:48]
Temp4[31:0] := SRC1[79:64] * SRC2[79:64]
Temp5[31:0] := SRC1[95:80] * SRC2[95:80]
Temp6[31:0] := SRC1[111:96] * SRC2[111:96]
Temp7[31:0] := SRC1[127:112] * SRC2[127:112]
DEST[15:0] := Temp0[15:0]
DEST[31:16] := Temp1[15:0]
DEST[47:32] := Temp2[15:0]
DEST[63:48] := Temp3[15:0]
DEST[79:64] := Temp4[15:0]
DEST[95:80] := Temp5[15:0]
DEST[111:96] := Temp6[15:0]
DEST[127:112] := Temp7[15:0]
DEST[MAXVL-1:128] := 0
```

PMULLW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN
            temp[31:0] := SRC1[i+15:i] * SRC2[i+15:i]
            DEST[i+15:i] := temp[15:0]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMULLW __m512i _mm512_mullo_epi16(__m512i a, __m512i b);
VPMULLW __m512i _mm512_mask_mullo_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULLW __m512i _mm512_maskz_mullo_epi16( __mmask32 k, __m512i a, __m512i b);
VPMULLW __m256i _mm256_mask_mullo_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULLW __m256i _mm256_maskz_mullo_epi16( __mmask16 k, __m256i a, __m256i b);
VPMULLW __m128i _mm_mask_mullo_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULLW __m128i _mm_maskz_mullo_epi16( __mmask8 k, __m128i a, __m128i b);
PMULLW __m64 _mm_mullo_pi16(__m64 m1, __m64 m2)
(V)PMULLW __m128i _mm_mullo_epi16 ( __m128i a, __m128i b)
VPMULLW __m256i _mm256_mullo_epi16 ( __m256i a, __m256i b);
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

PMULUDQ—Multiply Packed Unsigned Doubleword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F4 /r[1] PMULUDQ mm1, mm2/m64	A	V/V	SSE2	Multiply unsigned doubleword integer in mm1 by unsigned doubleword integer in mm2/m64, and store the quadword result in mm1.
66 0F F4 /r PMULUDQ xmm1, xmm2/m128	A	V/V	SSE2	Multiply packed unsigned doubleword integers in xmm1 by packed unsigned doubleword integers in xmm2/m128, and store the quadword results in xmm1.
VEX.128.66.0F.WIG F4 /r VPMULUDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply packed unsigned doubleword integers in xmm2 by packed unsigned doubleword integers in xmm3/m128, and store the quadword results in xmm1.
VEX.256.66.0F.WIG F4 /r VPMULUDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply packed unsigned doubleword integers in ymm2 by packed unsigned doubleword integers in ymm3/m256, and store the quadword results in ymm1.
EVEX.128.66.0F.W1 F4 /r VPMULUDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed unsigned doubleword integers in xmm2 by packed unsigned doubleword integers in xmm3/m128/m64bcst, and store the quadword results in xmm1 under writemask k1.
EVEX.256.66.0F.W1 F4 /r VPMULUDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed unsigned doubleword integers in ymm2 by packed unsigned doubleword integers in ymm3/m256/m64bcst, and store the quadword results in ymm1 under writemask k1.
EVEX.512.66.0F.W1 F4 /r VPMULUDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Multiply packed unsigned doubleword integers in zmm2 by packed unsigned doubleword integers in zmm3/m512/m64bcst, and store the quadword results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies the first operand (destination operand) by the second operand (source operand) and stores the result in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an unsigned doubleword integer stored in the low doubleword of an MMX technology register or a 64-bit memory location. The destination operand can be an unsigned doubleword integer stored in the low doubleword an MMX technology register. The result is an unsigned

quadword integer stored in the destination an MMX technology register. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

For 64-bit memory operands, 64 bits are fetched from memory, but only the low doubleword is used in the computation.

128-bit Legacy SSE version: The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand is two packed unsigned doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand is two packed unsigned doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is four packed unsigned doubleword integers stored in the first (low), third, fifth, and seventh doublewords of a YMM register or a 256-bit memory location. For 256-bit memory operands, 256 bits are fetched from memory, but only the first, third, fifth, and seventh doublewords are used in the computation. The first source operand is four packed unsigned doubleword integers stored in the first, third, fifth, and seventh doublewords of an YMM register. The destination contains four packed unaligned quadword integers stored in an YMM register.

EVEX encoded version: The input unsigned doubleword integers are taken from the even-numbered elements of the source operands. The first source operand is a ZMM/YMM/XMM registers. The second source operand can be an ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination is a ZMM/YMM/XMM register, and updated according to the writemask at 64-bit granularity.

Operation

PMULUDQ (With 64-Bit Operands)

```
DEST[63:0] := DEST[31:0] * SRC[31:0];
```

PMULUDQ (With 128-Bit Operands)

```
DEST[63:0] := DEST[31:0] * SRC[31:0];
DEST[127:64] := DEST[95:64] * SRC[95:64];
```

VPMULUDQ (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[31:0] * SRC2[31:0]
DEST[127:64] := SRC1[95:64] * SRC2[95:64]
DEST[MAXVL-1:128] := 0
```

VPMULUDQ (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[31:0] * SRC2[31:0]
DEST[127:64] := SRC1[95:64] * SRC2[95:64
DEST[191:128] := SRC1[159:128] * SRC2[159:128]
DEST[255:192] := SRC1[223:192] * SRC2[223:192]
DEST[MAXVL-1:256] := 0
```

VPMULUDQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := ZeroExtend64( SRC1[i+31:i]) * ZeroExtend64( SRC2[31:0] )
                ELSE DEST[i+63:i] := ZeroExtend64( SRC1[i+31:i]) * ZeroExtend64( SRC2[i+31:i] )
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMULUDQ __m512i _mm512_mul_epu32(__m512i a, __m512i b);
VPMULUDQ __m512i _mm512_mask_mul_epu32(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMULUDQ __m512i _mm512_maskz_mul_epu32( __mmask8 k, __m512i a, __m512i b);
VPMULUDQ __m256i _mm256_mask_mul_epu32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULUDQ __m256i _mm256_maskz_mul_epu32( __mmask8 k, __m256i a, __m256i b);
VPMULUDQ __m128i _mm_mask_mul_epu32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULUDQ __m128i _mm_maskz_mul_epu32( __mmask8 k, __m128i a, __m128i b);
PMULUDQ __m64 _mm_mul_su32 (__m64 a, __m64 b)
(V)PMULUDQ __m128i _mm_mul_epu32 ( __m128i a, __m128i b)
VPMULUDQ __m256i _mm256_mul_epu32( __m256i a, __m256i b);
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."

EVEX-encoded instruction, see Table 2-51, "Type E4 Class Exception Conditions."

POP—Pop a Value From the Stack

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
8F /0	POP r/m16	M	Valid	Valid	Pop top of stack into m16; increment stack pointer.
8F /0	POP r/m32	M	N.E.	Valid	Pop top of stack into m32; increment stack pointer.
8F /0	POP r/m64	M	Valid	N.E.	Pop top of stack into m64; increment stack pointer. Cannot encode 32-bit operand size.
58+ rw	POP r16	O	Valid	Valid	Pop top of stack into r16; increment stack pointer.
58+ rd	POP r32	O	N.E.	Valid	Pop top of stack into r32; increment stack pointer.
58+ rd	POP r64	O	Valid	N.E.	Pop top of stack into r64; increment stack pointer. Cannot encode 32-bit operand size.
1F	POP DS	ZO	Invalid	Valid	Pop top of stack into DS; increment stack pointer.
07	POP ES	ZO	Invalid	Valid	Pop top of stack into ES; increment stack pointer.
17	POP SS	ZO	Invalid	Valid	Pop top of stack into SS; increment stack pointer.
0F A1	POP FS	ZO	Valid	Valid	Pop top of stack into FS; increment stack pointer by 16 bits.
0F A1	POP FS	ZO	N.E.	Valid	Pop top of stack into FS; increment stack pointer by 32 bits.
0F A1	POP FS	ZO	Valid	N.E.	Pop top of stack into FS; increment stack pointer by 64 bits.
0F A9	POP GS	ZO	Valid	Valid	Pop top of stack into GS; increment stack pointer by 16 bits.
0F A9	POP GS	ZO	N.E.	Valid	Pop top of stack into GS; increment stack pointer by 32 bits.
0F A9	POP GS	ZO	Valid	N.E.	Pop top of stack into GS; increment stack pointer by 64 bits.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	N/A	N/A	N/A
O	opcode + rd (w)	N/A	N/A	N/A
ZO	N/A	N/A	N/A	N/A

Description

Loads the value from the top of the stack to the location specified with the destination operand (or explicit opcode) and then increments the stack pointer. The destination operand can be a general-purpose register, memory location, or segment register.

Address and operand sizes are determined and used as follows:

- Address size. The D flag in the current code-segment descriptor determines the default address size; it may be overridden by an instruction prefix (67H).

  The address size is used only when writing to a destination operand in memory.
- Operand size. The D flag in the current code-segment descriptor determines the default operand size; it may be overridden by instruction prefixes (66H or REX.W).

  The operand size (16, 32, or 64 bits) determines the amount by which the stack pointer is incremented (2, 4 or 8).
- Stack-address size. Outside of 64-bit mode, the B flag in the current stack-segment descriptor determines the size of the stack pointer (16 or 32 bits); in 64-bit mode, the size of the stack pointer is always 64 bits.

  The stack-address size determines the width of the stack pointer when reading from the stack in memory and when incrementing the stack pointer. (As stated above, the amount by which the stack pointer is incremented is determined by the operand size.)

If the destination operand is one of the segment registers DS, ES, FS, GS, or SS, the value loaded into the register must be a valid segment selector. In protected mode, popping a segment selector into a segment register automat-

ically causes the descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register and causes the selector and the descriptor information to be validated (see the "Operation" section below).

A NULL value (0000-0003) may be popped into the DS, ES, FS, or GS register without causing a general protection fault. However, any subsequent attempt to reference a segment whose corresponding segment register is loaded with a NULL value causes a general protection exception (#GP). In this situation, no memory reference occurs and the saved value of the segment register is NULL.

The POP instruction cannot pop a value into the CS register. To load the CS register from the stack, use the RET instruction.

If the ESP register is used as a base register for addressing a destination operand in memory, the POP instruction computes the effective address of the operand after it increments the ESP register. For the case of a 16-bit stack where ESP wraps to 0H as a result of the POP instruction, the resulting location of the memory write is processor-family-specific.

The POP ESP instruction increments the stack pointer (ESP) before data at the old top of stack is written into the destination.

Loading the SS register with a POP instruction suppresses or inhibits some debug exceptions and inhibits interrupts on the following instruction boundary. (The inhibition ends after delivery of an exception or the execution of the next instruction.) This behavior allows a stack pointer to be loaded into the ESP register with the next instruction (POP ESP) before an event can be delivered. See Section 7.8.3, "Masking Exceptions and Interrupts When Switching Stacks," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A. Intel recommends that software use the LSS instruction to load the SS register and ESP together.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). When in 64-bit mode, POPs using 32-bit operands are not encodable and POPs to DS, ES, SS are not valid. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF StackAddrSize = 32
    THEN
        IF OperandSize = 32
            THEN
                DEST := SS:ESP; (* Copy a doubleword *)
                ESP := ESP + 4;
            ELSE (* OperandSize = 16*)
                DEST := SS:ESP; (* Copy a word *)
                ESP := ESP + 2;
        FI;
    ELSE IF StackAddrSize = 64
        THEN
            IF OperandSize = 64
                THEN
                    DEST := SS:RSP; (* Copy quadword *)
                    RSP := RSP + 8;
                ELSE (* OperandSize = 16*)
                    DEST := SS:RSP; (* Copy a word *)
                    RSP := RSP + 2;
            FI;
        FI;
    ELSE StackAddrSize = 16
        THEN
            IF OperandSize = 16
                THEN
                    DEST := SS:SP; (* Copy a word *)
                    SP := SP + 2;
```

```
            ELSE (* OperandSize = 32 *)
                DEST := SS:SP; (* Copy a doubleword *)
                SP := SP + 4;
        FI;

FI;
```

Loading a segment register while in protected mode results in special actions, as described in the following listing. These checks are performed on the segment selector and the segment descriptor it points to.

```
64-BIT_MODE
IF FS, or GS is loaded with non-NULL selector;
    THEN
        IF segment selector index is outside descriptor table limits
            OR segment is not a data or readable code segment
            OR ((segment is a data or nonconforming code segment)
                AND ((RPL > DPL) or (CPL > DPL))
                    THEN #GP(selector);
            IF segment not marked present
                THEN #NP(selector);
        ELSE
            SegmentRegister := segment selector;
            SegmentRegister := segment descriptor;
        FI;
FI;
IF FS, or GS is loaded with a NULL selector;
        THEN
            SegmentRegister := segment selector;
            SegmentRegister := segment descriptor;
FI;


PREOTECTED MODE OR COMPATIBILITY MODE;
IF SS is loaded;
    THEN
        IF segment selector is NULL
            THEN #GP(0);
        FI;
        IF segment selector index is outside descriptor table limits
            or segment selector's RPL ≠ CPL
            or segment is not a writable data segment
            or DPL ≠ CPL
                THEN #GP(selector);
        FI;
        IF segment not marked present
            THEN #SS(selector);
            ELSE
                SS := segment selector;
                SS := segment descriptor;
        FI;
FI;

IF DS, ES, FS, or GS is loaded with non-NULL selector;
    THEN
```

```
        IF segment selector index is outside descriptor table limits
            or segment is not a data or readable code segment
            or ((segment is a data or nonconforming code segment)
            and ((RPL > DPL) or (CPL > DPL))
                THEN #GP(selector);
        FI;
        IF segment not marked present
            THEN #NP(selector);
            ELSE
                SegmentRegister := segment selector;
                SegmentRegister := segment descriptor;
         FI;
FI;

IF DS, ES, FS, or GS is loaded with a NULL selector
    THEN
        SegmentRegister := segment selector;
        SegmentRegister := segment descriptor;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If attempt is made to load SS register with NULL segment selector.
	If the destination operand is in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#GP(selector)	If segment selector index is outside descriptor table limits.
	If the SS register is being loaded and the segment selector's RPL and the segment descriptor's DPL are not equal to the CPL.
	If the SS register is being loaded and the segment pointed to is a non-writable data segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#SS(0)	If the current top of stack is not within the stack segment.
	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If the SS register is being loaded and the segment pointed to is marked not present.
#NP	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If the stack address is in a non-canonical form.
#GP(selector)	If the descriptor is outside the descriptor table limit.
	If the FS or GS register is being loaded and the segment pointed to is not a data or readable code segment.
	If the FS or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#PF(fault-code)	If a page fault occurs.
#NP	If the FS or GS register is being loaded and the segment pointed to is marked not present.
#UD	If the LOCK prefix is used.
	If the DS, ES, or SS register is being loaded.

POPA/POPAD—Pop All General-Purpose Registers

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
61	POPA	ZO	Invalid	Valid	Pop DI, SI, BP, BX, DX, CX, and AX.
61	POPAD	ZO	Invalid	Valid	Pop EDI, ESI, EBP, EBX, EDX, ECX, and EAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Pops doublewords (POPAD) or words (POPA) from the stack into the general-purpose registers. The registers are loaded in the following order: EDI, ESI, EBP, EBX, EDX, ECX, and EAX (if the operand-size attribute is 32) and DI, SI, BP, BX, DX, CX, and AX (if the operand-size attribute is 16). (These instructions reverse the operation of the PUSHA/PUSHAD instructions.) The value on the stack for the ESP or SP register is ignored. Instead, the ESP or SP register is incremented after each register is loaded.

The POPA (pop all) and POPAD (pop all double) mnemonics reference the same opcode. The POPA instruction is intended for use when the operand-size attribute is 16 and the POPAD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when POPA is used and to 32 when POPAD is used (using the operand-size override prefix [66H] if necessary). Others may treat these mnemonics as synonyms (POPA/POPAD) and use the current setting of the operand-size attribute to determine the size of values to be popped from the stack, regardless of the mnemonic used. (The D flag in the current code segment's segment descriptor determines the operand-size attribute.)

This instruction executes as described in non-64-bit modes. It is not valid in 64-bit mode.

Operation

```
IF 64-Bit Mode
    THEN
        #UD;
ELSE
    IF OperandSize = 32 (* Instruction = POPAD *)
    THEN
        EDI := Pop();
        ESI := Pop();
        EBP := Pop();
        Increment ESP by 4; (* Skip next 4 bytes of stack *)
        EBX := Pop();
        EDX := Pop();
        ECX := Pop();
        EAX := Pop();
    ELSE (* OperandSize = 16, instruction = POPA *)
        DI := Pop();
        SI := Pop();
        BP := Pop();
        Increment ESP by 2; (* Skip next 2 bytes of stack *)
        BX := Pop();
        DX := Pop();
        CX := Pop();
        AX := Pop();
    FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#SS(0)	If the starting or ending stack address is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#SS	If the starting or ending stack address is not within the stack segment.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#SS(0)	If the starting or ending stack address is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#UD	If in 64-bit mode.

POPCNT—Return the Count of Number of Bits Set to 1

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F B8 /r	POPCNT r16, r/m16	RM	Valid	Valid	POPCNT on r/m16
F3 0F B8 /r	POPCNT r32, r/m32	RM	Valid	Valid	POPCNT on r/m32
F3 REX.W 0F B8 /r	POPCNT r64, r/m64	RM	Valid	N.E.	POPCNT on r/m64

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction calculates the number of bits set to 1 in the second operand (source) and returns the count in the first operand (a destination register).

Operation

```
Count = 0;
For (i=0; i < OperandSize; i++)
{       IF (SRC[ i] = 1) // i'th bit
        THEN Count++; FI;
}
DEST := Count;
```

Flags Affected

OF, SF, ZF, AF, CF, PF are all cleared. ZF is set if SRC = 0, otherwise ZF is cleared.

Intel C/C++ Compiler Intrinsic Equivalent

POPCNT int _mm_popcnt_u32(unsigned int a);
POPCNT int64_t _mm_popcnt_u64(unsigned __int64 a);

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If CPUID.01H:ECX.POPCNT[23] = 0.
	If LOCK prefix is used.

Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CPUID.01H:ECX.POPCNT[23] = 0.
	If LOCK prefix is used.

Virtual 8086 Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If CPUID.01H:ECX.POPCNT[23] = 0.
	If LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If CPUID.01H:ECX.POPCNT[23] = 0.
	If LOCK prefix is used.

POPF/POPFD/POPFQ—Pop Stack Into EFLAGS Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
9D	POPF	ZO	Valid	Valid	Pop top of stack into lower 16 bits of EFLAGS.
9D	POPFD	ZO	N.E.	Valid	Pop top of stack into EFLAGS.
9D	POPFQ	ZO	Valid	N.E.	Pop top of stack and zero-extend into RFLAGS.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Pops a doubleword (POPFD) from the top of the stack (if the current operand-size attribute is 32) and stores the value in the EFLAGS register, or pops a word from the top of the stack (if the operand-size attribute is 16) and stores it in the lower 16 bits of the EFLAGS register (that is, the FLAGS register). These instructions reverse the operation of the PUSHF/PUSHFD/PUSHFQ instructions.

The POPF (pop flags) and POPFD (pop flags double) mnemonics reference the same opcode. The POPF instruction is intended for use when the operand-size attribute is 16; the POPFD instruction is intended for use when the operand-size attribute is 32. Some assemblers may force the operand size to 16 for POPF and to 32 for POPFD. Others may treat the mnemonics as synonyms (POPF/POPFD) and use the setting of the operand-size attribute to determine the size of values to pop from the stack.

The effect of POPF/POPFD on the EFLAGS register changes, depending on the mode of operation. See Table 4-20 and the key below for details.

When operating in protected, compatibility, or 64-bit mode at privilege level 0 (or in real-address mode, the equivalent to privilege level 0), all non-reserved flags in the EFLAGS register except RF[1], VIP, VIF, and VM may be modified. VIP, VIF, and VM remain unaffected.

When operating in protected, compatibility, or 64-bit mode with a privilege level greater than 0, but less than or equal to IOPL, all flags can be modified except the IOPL field and RF, IF, VIP, VIF, and VM; these remain unaffected. The AC and ID flags can only be modified if the operand-size attribute is 32. The interrupt flag (IF) is altered only when executing at a level at least as privileged as the IOPL. If a POPF/POPFD instruction is executed with insufficient privilege, an exception does not occur but privileged bits do not change.

When operating in virtual-8086 mode (EFLAGS.VM = 1) without the virtual-8086 mode extensions (CR4.VME = 0), the POPF/POPFD instructions can be used only if IOPL = 3; otherwise, a general-protection exception (#GP) occurs. If the virtual-8086 mode extensions are enabled (CR4.VME = 1), POPF (but not POPFD) can be executed in virtual-8086 mode with IOPL < 3.

(The protected-mode virtual-interrupt feature — enabled by setting CR4.PVI — affects the CLI and STI instructions in the same manner as the virtual-8086 mode extensions. POPF, however, is not affected by CR4.PVI.)

In 64-bit mode, the mnemonic assigned is POPFQ (note that the 32-bit operand is not encodable). POPFQ pops 64 bits from the stack. Reserved bits of RFLAGS (including the upper 32 bits of RFLAGS) are not affected.

See Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for more information about the EFLAGS registers.

1. RF is always zero after the execution of POPF. This is because POPF, like all instructions, clears RF as it begins to execute.

Table 4-20. Effect of POPF/POPFD on the EFLAGS Register

Mode	Operand Size	CPL	IOPL	Flags																Notes
				21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0	
				ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF	
Real-Address Mode (CR0.PE = 0)	16	0	0-3	N	N	N	N	N	0	S	S	S	S	S	S	S	S	S	S	S	
	32	0	0-3	S	N	N	S	N	0	S	S	S	S	S	S	S	S	S	S	S	
Protected, Compatibility, and 64-Bit Modes (CR0.PE = 1 EFLAGS.VM = 0)	16	0	0-3	N	N	N	N	N	0	S	S	S	S	S	S	S	S	S	S	S	
	16	1-3	<CPL	N	N	N	N	N	0	S	N	S	S	N	S	S	S	S	S	S	
	16	1-3	≥CPL	N	N	N	N	N	0	S	N	S	S	S	S	S	S	S	S	S	
	32, 64	0	0-3	S	N	N	S	N	0	S	S	S	S	S	S	S	S	S	S	S	
	32, 64	1-3	<CPL	S	N	N	S	N	0	S	N	S	S	N	S	S	S	S	S	S	
	32, 64	1-3	≥CPL	S	N	N	S	N	0	S	N	S	S	S	S	S	S	S	S	S	
Virtual-8086 (CR0.PE = 1 EFLAGS.VM = 1 CR4.VME = 0)	16	3	0-2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1
	16	3	3	N	N	N	N	N	0	S	N	S	S	S	S	S	S	S	S	S	
	32	3	0-2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1
	32	3	3	S	N	N	S	N	0	S	N	S	S	S	S	S	S	S	S	S	
VME (CR0.PE = 1 EFLAGS.VM = 1 CR4.VME = 1)	16	3	0-2	N/X	N/X	SV/X	N/X	N/X	0/X	S/X	N/X	S/X	S/X	N/X	S/X	S/X	S/X	S/X	S/X	S/X	2,3
	16	3	3	N	N	N	N	N	0	S	N	S	S	S	S	S	S	S	S	S	
	32	3	0-2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1
	32	3	3	S	N	N	S	N	0	S	N	S	S	S	S	S	S	S	S	S	

NOTES:

1. #GP fault - no flag update
2. #GP fault with no flag update if VIP=1 in EFLAGS register and IF=1 in FLAGS value on stack
3. #GP fault with no flag update if TF=1 in FLAGS value on stack

Key	
S	Updated from stack
SV	Updated from IF (bit 9) in FLAGS value on stack
N	No change in value
X	No EFLAGS update
0	Value is cleared

Operation

```
IF EFLAGS.VM = 0 (* Not in Virtual-8086 Mode *)
    THEN IF CPL = 0 OR CR0.PE = 0
        THEN
            IF OperandSize = 32;
                THEN
                    EFLAGS := Pop(); (* 32-bit pop *)
                    (* All non-reserved flags except RF, VIP, VIF, and VM can be modified;
                    VIP, VIF, VM, and all reserved bits are unaffected. RF is cleared. *)
                ELSE IF (Operandsize = 64)
                    RFLAGS = Pop(); (* 64-bit pop *)
                    (* All non-reserved flags except RF, VIP, VIF, and VM can be modified;
                    VIP, VIF, VM, and all reserved bits are unaffected. RF is cleared. *)
```

```
            ELSE (* OperandSize = 16 *)
                EFLAGS[15:0] := Pop(); (* 16-bit pop *)
                (* All non-reserved flags can be modified. *)
        FI;
    ELSE (* CPL > 0 *)
        IF OperandSize = 32
            THEN
                IF CPL > IOPL
                    THEN
                        EFLAGS := Pop(); (* 32-bit pop *)
                        (* All non-reserved bits except IF, IOPL, VIP, VIF, VM, and RF can be modified;
                        IF, IOPL, VIP, VIF, VM, and all reserved bits are unaffected; RF is cleared. *)
                    ELSE
                        EFLAGS := Pop(); (* 32-bit pop *)
                        (* All non-reserved bits except IOPL, VIP, VIF, VM, and RF can be modified;
                        IOPL, VIP, VIF, VM, and all reserved bits are unaffected; RF is cleared. *)
                FI;
            ELSE IF (Operandsize = 64)
                IF CPL > IOPL
                    THEN
                        RFLAGS := Pop(); (* 64-bit pop *)
                        (* All non-reserved bits except IF, IOPL, VIP, VIF, VM, and RF can be modified;
                        IF, IOPL, VIP, VIF, VM, and all reserved bits are unaffected; RF is cleared. *)
                    ELSE
                        RFLAGS := Pop(); (* 64-bit pop *)
                        (* All non-reserved bits except IOPL, VIP, VIF, VM, and RF can be modified;
                        IOPL, VIP, VIF, VM, and all reserved bits are unaffected; RF is cleared. *)
                FI;
            ELSE (* OperandSize = 16 *)
                EFLAGS[15:0] := Pop(); (* 16-bit pop *)
                (* All non-reserved bits except IOPL can be modified; IOPL and all
                reserved bits are unaffected. *)
        FI;
    FI;
ELSE (* In virtual-8086 mode *)
    IF IOPL = 3
        THEN
            IF OperandSize = 32
                THEN
                    EFLAGS := Pop();
                    (* All non-reserved bits except IOPL, VIP, VIF, VM, and RF can be modified;
                    VIP, VIF, VM, IOPL, and all reserved bits are unaffected. RF is cleared. *)
                ELSE
                    EFLAGS[15:0] := Pop(); FI;
                    (* All non-reserved bits except IOPL can be modified; IOPL and all reserved bits are unaffected. *)
            FI;
        ELSE (* IOPL < 3 *)
            IF (Operandsize = 32) OR (CR4.VME = 0)
                THEN #GP(0); (* Trap to virtual-8086 monitor. *)
                ELSE (* Operandsize = 16 and CR4.VME = 1 *)
                    tempFLAGS := Pop();
                    IF (EFLAGS.VIP = 1 AND tempFLAGS[9] = 1) OR tempFLAGS[8] = 1
                        THEN #GP(0);
                        ELSE
```

```
                                EFLAGS.VIF := tempFLAGS[9];
                                EFLAGS[15:0] := tempFLAGS;
                                (* All non-reserved bits except IOPL and IF can be modified;
                                IOPL, IF, and all reserved bits are unaffected. *)
                        FI;
                FI;
        FI;
FI;
```

Flags Affected

All flags may be affected; see the Operation section for details.

Protected Mode Exceptions

#SS(0)	If the top of stack is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while CPL = 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#SS	If the top of stack is not within the stack segment.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If IOPL < 3 and VME is not enabled.
	If IOPL < 3 and the 32-bit operand size is used.
	If IOPL < 3, EFLAGS.VIP = 1, and bit 9 (IF) is set in the FLAGS value on the stack.
	If IOPL < 3 and bit 8 (TF) is set in the FLAGS value on the stack.
	If an attempt is made to execute the POPF/POPFD instruction with an operand-size override prefix.
#SS(0)	If the top of stack is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

POR—Bitwise Logical OR

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EB /r[1] POR mm, mm/m64	A	V/V	MMX	Bitwise OR of mm/m64 and mm.
66 0F EB /r POR xmm1, xmm2/m128	A	V/V	SSE2	Bitwise OR of xmm2/m128 and xmm1.
VEX.128.66.0F.WIG EB /r VPOR xmm1, xmm2, xmm3/m128	B	V/V	AVX	Bitwise OR of xmm2/m128 and xmm3.
VEX.256.66.0F.WIG EB /r VPOR ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Bitwise OR of ymm2/m256 and ymm3.
EVEX.128.66.0F.W0 EB /r VPORD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise OR of packed doubleword integers in xmm2 and xmm3/m128/m32bcst using writemask k1.
EVEX.256.66.0F.W0 EB /r VPORD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise OR of packed doubleword integers in ymm2 and ymm3/m256/m32bcst using writemask k1.
EVEX.512.66.0F.W0 EB /r VPORD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Bitwise OR of packed doubleword integers in zmm2 and zmm3/m512/m32bcst using writemask k1.
EVEX.128.66.0F.W1 EB /r VPORQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise OR of packed quadword integers in xmm2 and xmm3/m128/m64bcst using writemask k1.
EVEX.256.66.0F.W1 EB /r VPORQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise OR of packed quadword integers in ymm2 and ymm3/m256/m64bcst using writemask k1.
EVEX.512.66.0F.W1 EB /r VPORQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Bitwise OR of packed quadword integers in zmm2 and zmm3/m512/m64bcst using writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical OR operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. Each bit of the result is set to 1 if either or both of the corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or a 256-bit memory location. The first source and destination operands can be YMM registers.

EVEX encoded version: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with write-mask k1 at 32/64-bit granularity.

Operation

POR (64-bit Operand)

```
DEST := DEST OR SRC
```

POR (128-bit Legacy SSE Version)

```
DEST := DEST OR SRC
DEST[MAXVL-1:128] (Unmodified)
```

VPOR (VEX.128 Encoded Version)

```
DEST := SRC1 OR SRC2
DEST[MAXVL-1:128] := 0
```

VPOR (VEX.256 Encoded Version)

```
DEST := SRC1 OR SRC2
DEST[MAXVL-1:256] := 0
```

VPORD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := SRC1[i+31:i] BITWISE OR SRC2[31:0]
                ELSE DEST[i+31:i] := SRC1[i+31:i] BITWISE OR SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPORD __m512i _mm512_or_epi32(__m512i a, __m512i b);
VPORD __m512i _mm512_mask_or_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPORD __m512i _mm512_maskz_or_epi32(__mmask16 k, __m512i a, __m512i b);
VPORD __m256i _mm256_or_epi32(__m256i a, __m256i b);
VPORD __m256i _mm256_mask_or_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b,);
VPORD __m256i _mm256_maskz_or_epi32(__mmask8 k, __m256i a, __m256i b);
VPORD __m128i _mm_or_epi32(__m128i a, __m128i b);
VPORD __m128i _mm_mask_or_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPORD __m128i _mm_maskz_or_epi32(__mmask8 k, __m128i a, __m128i b);
VPORQ __m512i _mm512_or_epi64(__m512i a, __m512i b);
VPORQ __m512i _mm512_mask_or_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPORQ __m512i _mm512_maskz_or_epi64(__mmask8 k, __m512i a, __m512i b);
VPORQ __m256i _mm256_or_epi64(__m256i a, int imm);
VPORQ __m256i _mm256_mask_or_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPORQ __m256i _mm256_maskz_or_epi64(__mmask8 k, __m256i a, __m256i b);
VPORQ __m128i _mm_or_epi64(__m128i a, __m128i b);
VPORQ __m128i _mm_mask_or_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPORQ __m128i _mm_maskz_or_epi64(__mmask8 k, __m128i a, __m128i b);
POR __m64 _mm_or_si64(__m64 m1, __m64 m2)
(V)POR __m128i _mm_or_si128(__m128i m1, __m128i m2)
VPOR __m256i _mm256_or_si256 (__m256i a, __m256i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

PREFETCH*h*—Prefetch Data Into Caches

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	Description
0F 18 /1 PREFETCHT0 *m8*	M	V/V	Move data from *m8* closer to the processor using T0 hint.
0F 18 /2 PREFETCHT1 *m8*	M	V/V	Move data from *m8* closer to the processor using T1 hint.
0F 18 /3 PREFETCHT2 *m8*	M	V/V	Move data from *m8* closer to the processor using T2 hint.
0F 18 /0 PREFETCHNTA *m8*	M	V/V	Move data from *m8* closer to the processor using NTA hint.
0F 18 /7 PREFETCHIT0 m8	M	V/I	Move code from relative address closer to the processor using IT0 hint.
0F 18 /6 PREFETCHIT1 m8	M	V/I	Move code from relative address closer to the processor using IT1 hint.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

Description

Fetches the line of data or code (instructions' bytes) from memory that contains the byte specified with the source operand to a location in the cache hierarchy specified by a locality hint:

- T0 (temporal data)—prefetch data into all levels of the cache hierarchy.
- T1 (temporal data with respect to first level cache misses)—prefetch data into level 2 cache and higher.
- T2 (temporal data with respect to second level cache misses)—prefetch data into level 3 cache and higher, or an implementation-specific choice.
- NTA (non-temporal data with respect to all cache levels)—prefetch data into non-temporal cache structure and into a location close to the processor, minimizing cache pollution.
- IT0 (temporal code)—prefetch code into all levels of the cache hierarchy.
- IT1 (temporal code with respect to first level cache misses)—prefetch code into all but the first-level of the cache hierarchy.

The source operand is a byte memory location. (The locality hints are encoded into the machine level instruction using bits 3 through 5 of the ModR/M byte.) Some locality hints may prefetch only for RIP-relative memory addresses; see additional details below. The address to prefetch is NextRIP + 32-bit displacement, where NextRIP is the first byte of the instruction that follows the prefetch instruction itself.

If the line selected is already present in the cache hierarchy at a level closer to the processor, no data movement occurs. Prefetches from uncacheable or WC memory are ignored.

The PREFETCH*h* instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor in anticipation of future use.

The implementation of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data or code lines prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes. Additional details of the implementation-dependent locality hints are described in Section 7.4 of Intel® 64 and IA-32 Architectures Optimization Reference Manual.

It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type that permits speculative reads (that is, the WB, WC, and WT memory types). A

PREFETCH*h* instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, a PREFETCH*h* instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCH*h* instruction is also unordered with respect to CLFLUSH and CLFLUSHOPT instructions, other PREFETCH*h* instructions, or any other general instruction. It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

PREFETCHIT0/1 can be used in 64-bit mode with RIP-relative addressing; they remain NOPs otherwise. For optimal performance, the addresses used with these instructions should be the starting byte of a real instruction.

PREFETCHIT0/1 instructions are enumerated by CPUID.07H.01H:EDX.PREFETCHI[14].The encodings remain NOPs in processors that do not enumerate these instructions.

Operation

```
FETCH (m8);
```

Intel C/C++ Compiler Intrinsic Equivalent

void _mm_prefetch(char *p, int i)
The argument "*p" gives the address of the byte (and corresponding cache line) to be prefetched. The value "i" gives a constant (_MM_HINT_T0, _MM_HINT_T1, _MM_HINT_T2, _MM_HINT_NTA, _MM_HINT_IT0, or _MM_HINT_IT1) that specifies the type of prefetch operation to be performed.

Numeric Exceptions

None.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

PREFETCHW—Prefetch Data Into Caches in Anticipation of a Write

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 0D /1 PREFETCHW m8	M	V/V	PREFETCHW	Move data from m8 closer to the processor in anticipation of a write.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

Description

Fetches the cache line of data from memory that contains the byte specified with the source operand to a location in the 1st or 2nd level cache and invalidates other cached instances of the line.

The source operand is a byte memory location. If the line selected is already present in the lowest level cache and is already in an exclusively owned state, no data movement occurs. Prefetches from non-writeback memory are ignored.

The PREFETCHW instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor and invalidates other cached copies in anticipation of the line being written to in the future.

The characteristic of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes. Additional details of the implementation-dependent locality hints are described in Section 7.4 of Intel® 64 and IA-32 Architectures Optimization Reference Manual.

It should be noted that processors are free to speculatively fetch and cache data with exclusive ownership from system memory regions that permit such accesses (that is, the WB memory type). A PREFETCHW instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, a PREFETCHW instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCHW instruction is also unordered with respect to CLFLUSH and CLFLUSHOPT instructions, other PREFETCHW instructions, or any other general instruction

It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
FETCH_WITH_EXCLUSIVE_OWNERSHIP (m8);
```

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

```
void _m_prefetchw( void * );
```

Protected Mode Exceptions

#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.

PSADBW—Compute Sum of Absolute Differences

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F6 /r[1] PSADBW mm1, mm2/m64	A	V/V	SSE	Computes the absolute differences of the packed unsigned byte integers from mm2 /m64 and mm1; differences are then summed to produce an unsigned word integer result.
66 0F F6 /r PSADBW xmm1, xmm2/m128	A	V/V	SSE2	Computes the absolute differences of the packed unsigned byte integers from xmm2 /m128 and xmm1; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.
VEX.128.66.0F.WIG F6 /r VPSADBW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Computes the absolute differences of the packed unsigned byte integers from xmm3 /m128 and xmm2; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.
VEX.256.66.0F.WIG F6 /r VPSADBW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Computes the absolute differences of the packed unsigned byte integers from ymm3 /m256 and ymm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.
EVEX.128.66.0F.WIG F6 /r VPSADBW xmm1, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Computes the absolute differences of the packed unsigned byte integers from xmm3 /m128 and xmm2; then each consecutive 8 differences are summed separately to produce two unsigned word integer results.
EVEX.256.66.0F.WIG F6 /r VPSADBW ymm1, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Computes the absolute differences of the packed unsigned byte integers from ymm3 /m256 and ymm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.
EVEX.512.66.0F.WIG F6 /r VPSADBW zmm1, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Computes the absolute differences of the packed unsigned byte integers from zmm3 /m512 and zmm2; then each consecutive 8 differences are summed separately to produce eight unsigned word integer results.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	N/A

Description

Computes the absolute value of the difference of 8 unsigned byte integers from the source operand (second operand) and from the destination operand (first operand). These 8 differences are then summed to produce an unsigned word integer result that is stored in the destination operand. Figure 4-14 shows the operation of the PSADBW instruction when using 64-bit operands.

When operating on 64-bit operands, the word integer result is stored in the low word of the destination operand, and the remaining bytes in the destination operand are cleared to all 0s.

When operating on 128-bit operands, two packed results are computed. Here, the 8 low-order bytes of the source and destination operands are operated on to produce a word result that is stored in the low word of the destination operand, and the 8 high-order bytes are operated on to produce a word result that is stored in bits 64 through 79 of the destination operand. The remaining bytes of the destination operand are cleared.

For 256-bit version, the third group of 8 differences are summed to produce an unsigned word in bits[143:128] of the destination register and the fourth group of 8 differences are summed to produce an unsigned word in bits[207:192] of the destination register. The remaining words of the destination are set to 0.

For 512-bit version, the fifth group result is stored in bits [271:256] of the destination. The result from the sixth group is stored in bits [335:320]. The results for the seventh and eighth group are stored respectively in bits [399:384] and bits [463:447], respectively. The remaining bits in the destination are set to 0.

In 64-bit mode and not encoded by VEX/EVEX prefix, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source operand and destination register are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: The first source operand and destination register are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 and EVEX.256 encoded versions: The first source operand and destination register are YMM registers. The second source operand is an YMM register or a 256-bit memory location. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The first source operand and destination register are ZMM registers. The second source operand is a ZMM register or a 512-bit memory location.

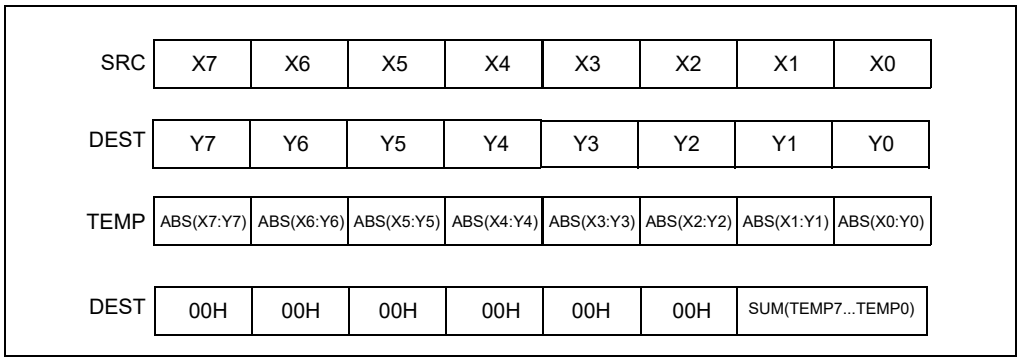


Figure 4-14. PSADBW Instruction Operation Using 64-bit Operands

Operation

VPSADBW (EVEX Encoded Versions)

```
VL = 128, 256, 512
TEMP0 := ABS(SRC1[7:0] - SRC2[7:0])
(* Repeat operation for bytes 1 through 15 *)
TEMP15 := ABS(SRC1[127:120] - SRC2[127:120])
DEST[15:0] := SUM(TEMP0:TEMP7)
DEST[63:16] := 000000000000H
DEST[79:64] := SUM(TEMP8:TEMP15)
DEST[127:80] := 00000000000H

IF VL >= 256
    (* Repeat operation for bytes 16 through 31*)
    TEMP31 := ABS(SRC1[255:248] - SRC2[255:248])
    DEST[143:128] := SUM(TEMP16:TEMP23)
    DEST[191:144] := 000000000000H
    DEST[207:192] := SUM(TEMP24:TEMP31)
    DEST[223:208] := 00000000000H
FI;
IF VL >= 512
(* Repeat operation for bytes 32 through 63*)
    TEMP63 := ABS(SRC1[511:504] - SRC2[511:504])
    DEST[271:256] := SUM(TEMP0:TEMP7)
    DEST[319:272] := 000000000000H
    DEST[335:320] := SUM(TEMP8:TEMP15)
    DEST[383:336] := 00000000000H
    DEST[399:384] := SUM(TEMP16:TEMP23)
    DEST[447:400] := 000000000000H
    DEST[463:448] := SUM(TEMP24:TEMP31)
    DEST[511:464] := 00000000000H
FI;
DEST[MAXVL-1:VL] := 0
```

VPSADBW (VEX.256 Encoded Version)

```
TEMP0 := ABS(SRC1[7:0] - SRC2[7:0])
(* Repeat operation for bytes 2 through 30*)
TEMP31 := ABS(SRC1[255:248] - SRC2[255:248])
DEST[15:0] := SUM(TEMP0:TEMP7)
DEST[63:16] := 000000000000H
DEST[79:64] := SUM(TEMP8:TEMP15)
DEST[127:80] := 00000000000H
DEST[143:128] := SUM(TEMP16:TEMP23)
DEST[191:144] := 000000000000H
DEST[207:192] := SUM(TEMP24:TEMP31)
DEST[223:208] := 00000000000H
DEST[MAXVL-1:256] := 0
```

VPSADBW (VEX.128 Encoded Version)
TEMP0 := ABS(SRC1[7:0] - SRC2[7:0])
(* Repeat operation for bytes 2 through 14 *)
TEMP15 := ABS(SRC1[127:120] - SRC2[127:120])
DEST[15:0] := SUM(TEMP0:TEMP7)
DEST[63:16] := 000000000000H
DEST[79:64] := SUM(TEMP8:TEMP15)
DEST[127:80] := 00000000000H
DEST[MAXVL-1:128] := 0

PSADBW (128-bit Legacy SSE Version)
TEMP0 := ABS(DEST[7:0] - SRC[7:0])
(* Repeat operation for bytes 2 through 14 *)
TEMP15 := ABS(DEST[127:120] - SRC[127:120])
DEST[15:0] := SUM(TEMP0:TEMP7)
DEST[63:16] := 000000000000H
DEST[79:64] := SUM(TEMP8:TEMP15)
DEST[127:80] := 00000000000
DEST[MAXVL-1:128] (Unmodified)

PSADBW (64-bit Operand)
TEMP0 := ABS(DEST[7:0] - SRC[7:0])
(* Repeat operation for bytes 2 through 6 *)
TEMP7 := ABS(DEST[63:56] - SRC[63:56])
DEST[15:0] := SUM(TEMP0:TEMP7)
DEST[63:16] := 000000000000H

Intel C/C++ Compiler Intrinsic Equivalent

VPSADBW __m512i _mm512_sad_epu8(__m512i a, __m512i b)
PSADBW __m64 _mm_sad_pu8(__m64 a,__m64 b)
(V)PSADBW __m128i _mm_sad_epu8(__m128i a, __m128i b)
VPSADBW __m256i _mm256_sad_epu8(__m256i a, __m256i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-52, “Type E4NF Class Exception Conditions.”

PSHUFB—Packed Shuffle Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 00 /r[1] PSHUFB mm1, mm2/m64	A	V/V	SSSE3	Shuffle bytes in mm1 according to contents of mm2/m64.
66 0F 38 00 /r PSHUFB xmm1, xmm2/m128	A	V/V	SSSE3	Shuffle bytes in xmm1 according to contents of xmm2/m128.
VEX.128.66.0F38.WIG 00 /r VPSHUFB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Shuffle bytes from xmm2 into xmm1 according to contents of xmm3/m128.
VEX.256.66.0F38.WIG 00 /r VPSHUFB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Shuffle bytes from ymm2 into ymm1 according to contents of ymm3/m256.
EVEX.128.66.0F38.WIG 00 /r VPSHUFB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shuffle bytes from xmm2 into xmm1 according to contents of xmm3/m128 under write mask k1.
EVEX.256.66.0F38.WIG 00 /r VPSHUFB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shuffle bytes from ymm2 into ymm1 according to contents of ymm3/m256 under write mask k1.
EVEX.512.66.0F38.WIG 00 /r VPSHUFB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Shuffle bytes from zmm2 into zmm1 according to contents of zmm3/m512 under write mask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

PSHUFB (with no VEX or EVEX prefix) performs an in-place shuffle of bytes in the destination operand (the first operand) according to the shuffle control mask in the source operand (the second operand). The instruction permutes the data in the destination operand, leaving the shuffle mask unaffected. If the most significant bit (bit[7]) of each byte of the shuffle control mask is set, then constant zero is written in the result byte. Each byte in the shuffle control mask forms an index to permute the corresponding byte in the destination operand. The value of each index is the least significant 3 bits (64-bit operation) or 4 bits (128-bit operation) of the shuffle control byte. See Figure 4-15 for an example for 64-bit operation.

The 128-bit forms of PSHUFB leave bits MAXVL–1:128 of the destination register unchanged. A 128-bit memory operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. In 64-bit mode, the REX prefix can be used to access XMM8-XMM15.

The following items apply to VPSHUFB, encoded with a VEX or EVEX prefix:

- Each of these forms uses shuffle control bytes in its second source operand to select which bytes in the first source operand to copy to the destination operand.

- These forms operate in one, two, or four 16-byte "lanes." As with the 128-bit form of PSHUFB (above), the low 4 bits of each shuffle control byte determines which of 16 bytes in a source lane is copied to the appropriate byte in the corresponding destination lane.
- The 128-bit and 256-bit versions of these forms zero upper bits in the destination register beyond the instruction's operand size.
- The EVEX-encoded versions update their destination conditionally with writemask k1.

Operation

PSHUFB (with 64-bit MMX operands)

```
TEMP := DEST
FOR destpos := 0 TO 7
    shufbyte := SRC.byte[destpos];
    IF shufbyte & 80H = 80H
        THEN DEST.byte[destpos] := 0;
        ELSE
            srcpos := shufbyte & 07H;
            DEST.byte[destpos] := TEMP.byte[srcpos];
    FI;
```

PSHUFB (with 128-bit SSE operands)

```
TEMP := DEST;
FOR destpos := 0 TO 15
    shufbyte := SRC.byte[destpos];
    IF shufbyte & 80H = 80H
        THEN DEST.byte[destpos] := 0;
        ELSE
            srcpos := shufbyte & 0FH;
            DEST.byte[destpos] := TEMP.byte[srcpos];
    FI;
```

VPSHUFB (VEX.128 encoded version)

```
FOR destpos := 0 TO 15
    shufbyte := SRC2.byte[destpos];
    IF shufbyte & 80H = 80H
        THEN DEST.byte[destpos] := 0;
        ELSE
            srcpos := shufbyte & 0FH;
            DEST.byte[destpos] := SRC1.byte[srcpos];
    FI;
DEST[MAXVL-1:128] := 0;
```

VPSHUFB (VEX.256 encoded version)

```
FOR lane := 0 to 1
    FOR lanepos := 0 TO 15
        destpos := 16 * lane + lanepos;
        shufbyte := SRC2.byte[destpos];
        IF shufbyte & 80H = 80H
            THEN DEST.byte[destpos] := 0;
            ELSE
                srcpos := 16 * lane + (shufbyte & 0FH);
                DEST.byte[destpos] := SRC1.byte[srcpos];
        FI;
DEST[MAXVL-1:256] := 0;
```

VPSHUFB (EVEX encoded versions)

```
// VL is 128, 256, or 512, depending on instruction encoding
// no masking if EVEX.aaa = 0; zeroing if EVEX.z = 1
FOR lane := 0 to VL/128 – 1
    FOR lanepos := 0 TO 15
        destpos := 16 * lane + lanepos;
        IF no masking OR k[destpos] = 1        // using selected bit from k register
            THEN
                shufbyte := SRC2.byte[destpos];
                IF shufbyte & 80H = 80H
                    THEN DEST.byte[destpos] := 0;
                    ELSE
                        srcpos := 16 * lane + (shufbyte & 0FH);
                        DEST.byte[destpos] := SRC1.byte[srcpos];
                FI;
            ELSE IF zeroing                    // if not zeroing, DEST.byte[destpos] is unchanged
                THEN DEST.byte[destpos] := 0;
        FI;
DEST[MAXVL–1:VL] := 0;
```

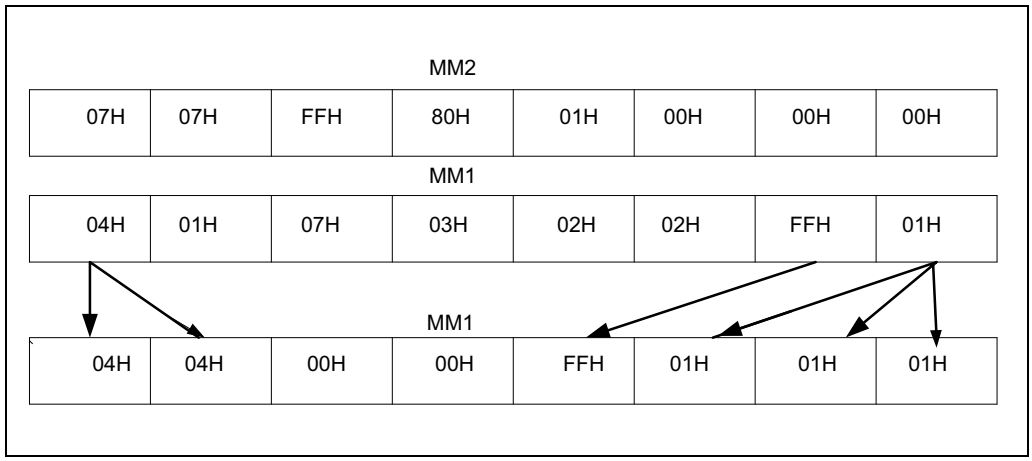


Figure 4-15. PSHUFB with 64-Bit Operands

Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFB __m512i _mm512_shuffle_epi8(__m512i a, __m512i b);
VPSHUFB __m512i _mm512_mask_shuffle_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSHUFB __m512i _mm512_maskz_shuffle_epi8(__mmask64 k, __m512i a, __m512i b);
VPSHUFB __m256i _mm256_mask_shuffle_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSHUFB __m256i _mm256_maskz_shuffle_epi8(__mmask32 k, __m256i a, __m256i b);
VPSHUFB __m128i _mm_mask_shuffle_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSHUFB __m128i _mm_maskz_shuffle_epi8(__mmask16 k, __m128i a, __m128i b);
PSHUFB: __m64 _mm_shuffle_pi8 (__m64 a, __m64 b)
(V)PSHUFB: __m128i _mm_shuffle_epi8 (__m128i a, __m128i b)
VPSHUFB:__m256i _mm256_shuffle_epi8(__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-52, “Type E4NF Class Exception Conditions.”

PSHUFD—Shuffle Packed Doublewords

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 70 /r ib PSHUFD xmm1, xmm2/m128, imm8	A	V/V	SSE2	Shuffle the doublewords in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.128.66.0F.WIG 70 /r ib VPSHUFD xmm1, xmm2/m128, imm8	A	V/V	AVX	Shuffle the doublewords in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.256.66.0F.WIG 70 /r ib VPSHUFD ymm1, ymm2/m256, imm8	A	V/V	AVX2	Shuffle the doublewords in ymm2/m256 based on the encoding in imm8 and store the result in ymm1.
EVEX.128.66.0F.W0 70 /r ib VPSHUFD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shuffle the doublewords in xmm2/m128/m32bcst based on the encoding in imm8 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F.W0 70 /r ib VPSHUFD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shuffle the doublewords in ymm2/m256/m32bcst based on the encoding in imm8 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F.W0 70 /r ib VPSHUFD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	B	V/V	AVX512F OR AVX10.1	Shuffle the doublewords in zmm2/m512/m32bcst based on the encoding in imm8 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Copies doublewords from source operand (second operand) and inserts them in the destination operand (first operand) at the locations selected with the order operand (third operand). Figure 4-16 shows the operation of the 256-bit VPSHUFD instruction and the encoding of the order operand. Each 2-bit field in the order operand selects the contents of one doubleword location within a 128-bit lane and copy to the target element in the destination operand. For example, bits 0 and 1 of the order operand targets the first doubleword element in the low and high 128-bit lane of the destination operand for 256-bit VPSHUFD. The encoded value of bits 1:0 of the order operand (see the field encoding in Figure 4-16) determines which doubleword element (from the respective 128-bit lane) of the source operand will be copied to doubleword 0 of the destination operand.

For 128-bit operation, only the low 128-bit lane are operative. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

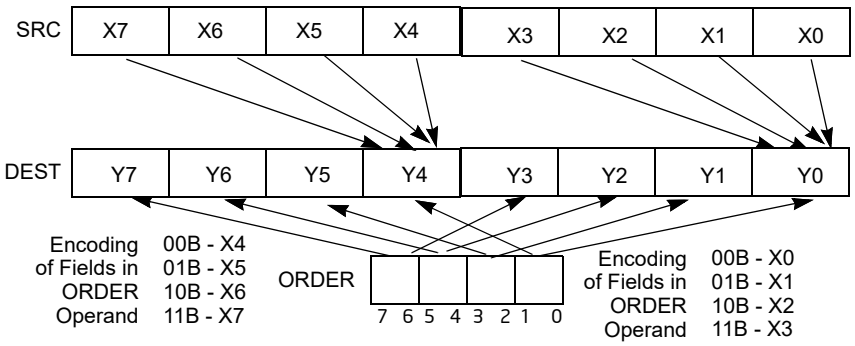


Figure 4-16. 256-bit VPSHUFD Instruction Operation

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

In 64-bit mode and not encoded in VEX/EVEX, using REX.R permits this instruction to access XMM8-XMM15.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 encoded version: The source operand can be an YMM register or a 256-bit memory location. The destination operand is an YMM register. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed. Bits (255-1:128) of the destination stores the shuffled results of the upper 16 bytes of the source operand using the immediate byte as the order operand.

EVEX encoded version: The source operand can be an ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

Each 128-bit lane of the destination stores the shuffled results of the respective lane of the source operand using the immediate byte as the order operand.

Note: EVEX.vvvv and VEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

PSHUFD (128-bit Legacy SSE Version)

```
DEST[31:0] := (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] := (SRC >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] := (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] := (SRC >> (ORDER[7:6] * 32))[31:0];
DEST[MAXVL-1:128] (Unmodified)
```

VPSHUFD (VEX.128 Encoded Version)

```
DEST[31:0] := (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] := (SRC >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] := (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] := (SRC >> (ORDER[7:6] * 32))[31:0];
DEST[MAXVL-1:128] := 0
```

VPSHUFD (VEX.256 Encoded Version)

```
DEST[31:0] := (SRC[127:0] >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] := (SRC[127:0] >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] := (SRC[127:0] >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] := (SRC[127:0] >> (ORDER[7:6] * 32))[31:0];
DEST[159:128] := (SRC[255:128] >> (ORDER[1:0] * 32))[31:0];
DEST[191:160] := (SRC[255:128] >> (ORDER[3:2] * 32))[31:0];
DEST[223:192] := (SRC[255:128] >> (ORDER[5:4] * 32))[31:0];
DEST[255:224] := (SRC[255:128] >> (ORDER[7:6] * 32))[31:0];
DEST[MAXVL-1:256] := 0
```

VPSHUFD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1) AND (SRC *is memory*)
        THEN TMP_SRC[i+31:i] := SRC[31:0]
        ELSE TMP_SRC[i+31:i] := SRC[i+31:i]
    FI;
ENDFOR;
IF VL >= 128
    TMP_DEST[31:0] := (TMP_SRC[127:0] >> (ORDER[1:0] * 32))[31:0];
    TMP_DEST[63:32] := (TMP_SRC[127:0] >> (ORDER[3:2] * 32))[31:0];
    TMP_DEST[95:64] := (TMP_SRC[127:0] >> (ORDER[5:4] * 32))[31:0];
    TMP_DEST[127:96] := (TMP_SRC[127:0] >> (ORDER[7:6] * 32))[31:0];
FI;
IF VL >= 256
    TMP_DEST[159:128] := (TMP_SRC[255:128] >> (ORDER[1:0] * 32))[31:0];
    TMP_DEST[191:160] := (TMP_SRC[255:128] >> (ORDER[3:2] * 32))[31:0];
    TMP_DEST[223:192] := (TMP_SRC[255:128] >> (ORDER[5:4] * 32))[31:0];
    TMP_DEST[255:224] := (TMP_SRC[255:128] >> (ORDER[7:6] * 32))[31:0];
FI;
IF VL >= 512
    TMP_DEST[287:256] := (TMP_SRC[383:256] >> (ORDER[1:0] * 32))[31:0];
    TMP_DEST[319:288] := (TMP_SRC[383:256] >> (ORDER[3:2] * 32))[31:0];
    TMP_DEST[351:320] := (TMP_SRC[383:256] >> (ORDER[5:4] * 32))[31:0];
    TMP_DEST[383:352] := (TMP_SRC[383:256] >> (ORDER[7:6] * 32))[31:0];
    TMP_DEST[415:384] := (TMP_SRC[511:384] >> (ORDER[1:0] * 32))[31:0];
    TMP_DEST[447:416] := (TMP_SRC[511:384] >> (ORDER[3:2] * 32))[31:0];
    TMP_DEST[479:448] := (TMP_SRC[511:384] >> (ORDER[5:4] * 32))[31:0];
    TMP_DEST[511:480] := (TMP_SRC[511:384] >> (ORDER[7:6] * 32))[31:0];
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
```

```
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFD __m512i _mm512_shuffle_epi32(__m512i a, int n);
VPSHUFD __m512i _mm512_mask_shuffle_epi32(__m512i s, __mmask16 k, __m512i a, int n);
VPSHUFD __m512i _mm512_maskz_shuffle_epi32(__mmask16 k, __m512i a, int n);
VPSHUFD __m256i _mm256_mask_shuffle_epi32(__m256i s, __mmask8 k, __m256i a, int n);
VPSHUFD __m256i _mm256_maskz_shuffle_epi32(__mmask8 k, __m256i a, int n);
VPSHUFD __m128i _mm_mask_shuffle_epi32(__m128i s, __mmask8 k, __m128i a, int n);
VPSHUFD __m128i _mm_maskz_shuffle_epi32(__mmask8 k, __m128i a, int n);
(V)PSHUFD __m128i _mm_shuffle_epi32(__m128i a, int n)
VPSHUFD __m256i _mm256_shuffle_epi32(__m256i a, const int n)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-52, "Type E4NF Class Exception Conditions."
Additionally:

#UD	If VEX.vvvv ≠ 1111B or EVEX.vvvv ≠ 1111B.

PSHUFHW—Shuffle Packed High Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 70 /r ib PSHUFHW xmm1, xmm2/m128, imm8	A	V/V	SSE2	Shuffle the high words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.128.F3.0F.WIG 70 /r ib VPSHUFHW xmm1, xmm2/m128, imm8	A	V/V	AVX	Shuffle the high words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.256.F3.0F.WIG 70 /r ib VPSHUFHW ymm1, ymm2/m256, imm8	A	V/V	AVX2	Shuffle the high words in ymm2/m256 based on the encoding in imm8 and store the result in ymm1.
EVEX.128.F3.0F.WIG 70 /r ib VPSHUFHW xmm1 {k1}{z}, xmm2/m128, imm8	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shuffle the high words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1 under write mask k1.
EVEX.256.F3.0F.WIG 70 /r ib VPSHUFHW ymm1 {k1}{z}, ymm2/m256, imm8	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shuffle the high words in ymm2/m256 based on the encoding in imm8 and store the result in ymm1 under write mask k1.
EVEX.512.F3.0F.WIG 70 /r ib VPSHUFHW zmm1 {k1}{z}, zmm2/m512, imm8	B	V/V	AVX512BW OR AVX10.1	Shuffle the high words in zmm2/m512 based on the encoding in imm8 and store the result in zmm1 under write mask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
B	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Copies words from the high quadword of a 128-bit lane of the source operand and inserts them in the high quadword of the destination operand at word locations (of the respective lane) selected with the immediate operand. This 256-bit operation is similar to the in-lane operation used by the 256-bit VPSHUFD instruction, which is illustrated in Figure 4-16. For 128-bit operation, only the low 128-bit lane is operative. Each 2-bit field in the immediate operand selects the contents of one word location in the high quadword of the destination operand. The binary encodings of the immediate operand fields select words (0, 1, 2 or 3, 4) from the high quadword of the source operand to be copied to the destination operand. The low quadword of the source operand is copied to the low quadword of the destination operand, for each 128-bit lane.

Note that this instruction permits a word in the high quadword of the source operand to be copied to more than one word location in the high quadword of the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The destination operand is an YMM register. The source operand can be an YMM register or a 256-bit memory location.

EVEX encoded version: The destination operand is a ZMM/YMM/XMM registers. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the write-mask.

Note: In VEX encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

PSHUFHW (128-bit Legacy SSE Version)

```
DEST[63:0] := SRC[63:0]
DEST[79:64] := (SRC >> (imm[1:0] *16))[79:64]
DEST[95:80] := (SRC >> (imm[3:2] * 16))[79:64]
DEST[111:96] := (SRC >> (imm[5:4] * 16))[79:64]
DEST[127:112] := (SRC >> (imm[7:6] * 16))[79:64]
DEST[MAXVL-1:128] (Unmodified)
```

VPSHUFHW (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0]
DEST[79:64] := (SRC1 >> (imm[1:0] *16))[79:64]
DEST[95:80] := (SRC1 >> (imm[3:2] * 16))[79:64]
DEST[111:96] := (SRC1 >> (imm[5:4] * 16))[79:64]
DEST[127:112] := (SRC1 >> (imm[7:6] * 16))[79:64]
DEST[MAXVL-1:128] := 0
```

VPSHUFHW (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0]
DEST[79:64] := (SRC1 >> (imm[1:0] *16))[79:64]
DEST[95:80] := (SRC1 >> (imm[3:2] * 16))[79:64]
DEST[111:96] := (SRC1 >> (imm[5:4] * 16))[79:64]
DEST[127:112] := (SRC1 >> (imm[7:6] * 16))[79:64]
DEST[191:128] := SRC1[191:128]
DEST[207192] := (SRC1 >> (imm[1:0] *16))[207:192]
DEST[223:208] := (SRC1 >> (imm[3:2] * 16))[207:192]
DEST[239:224] := (SRC1 >> (imm[5:4] * 16))[207:192]
DEST[255:240] := (SRC1 >> (imm[7:6] * 16))[207:192]
DEST[MAXVL-1:256] := 0
```

VPSHUFHW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL >= 128
    TMP_DEST[63:0] := SRC1[63:0]
    TMP_DEST[79:64] := (SRC1 >> (imm[1:0] *16))[79:64]
    TMP_DEST[95:80] := (SRC1 >> (imm[3:2] * 16))[79:64]
    TMP_DEST[111:96] := (SRC1 >> (imm[5:4] * 16))[79:64]
    TMP_DEST[127:112] := (SRC1 >> (imm[7:6] * 16))[79:64]
FI;
IF VL >= 256
    TMP_DEST[191:128] := SRC1[191:128]
    TMP_DEST[207:192] := (SRC1 >> (imm[1:0] *16))[207:192]
    TMP_DEST[223:208] := (SRC1 >> (imm[3:2] * 16))[207:192]
    TMP_DEST[239:224] := (SRC1 >> (imm[5:4] * 16))[207:192]
    TMP_DEST[255:240] := (SRC1 >> (imm[7:6] * 16))[207:192]
FI;
IF VL >= 512
    TMP_DEST[319:256] := SRC1[319:256]
    TMP_DEST[335:320] := (SRC1 >> (imm[1:0] *16))[335:320]
```

```
    TMP_DEST[351:336] := (SRC1 >> (imm[3:2] * 16))[335:320]
    TMP_DEST[367:352] := (SRC1 >> (imm[5:4] * 16))[335:320]
    TMP_DEST[383:368] := (SRC1 >> (imm[7:6] * 16))[335:320]
    TMP_DEST[447:384] := SRC1[447:384]
    TMP_DEST[463:448] := (SRC1 >> (imm[1:0] *16))[463:448]
    TMP_DEST[479:464] := (SRC1 >> (imm[3:2] * 16))[463:448]
    TMP_DEST[495:480] := (SRC1 >> (imm[5:4] * 16))[463:448]
    TMP_DEST[511:496] := (SRC1 >> (imm[7:6] * 16))[463:448]
FI;

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TMP_DEST[i+15:i];
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPSHUFHW __m512i _mm512_shufflehi_epi16(__m512i a, int n);
VPSHUFHW __m512i _mm512_mask_shufflehi_epi16(__m512i s, __mmask16 k, __m512i a, int n );
VPSHUFHW __m512i _mm512_maskz_shufflehi_epi16( __mmask16 k, __m512i a, int n );
VPSHUFHW __m256i _mm256_mask_shufflehi_epi16(__m256i s, __mmask8 k, __m256i a, int n );
VPSHUFHW __m256i _mm256_maskz_shufflehi_epi16( __mmask8 k, __m256i a, int n );
VPSHUFHW __m128i _mm_mask_shufflehi_epi16(__m128i s, __mmask8 k, __m128i a, int n );
VPSHUFHW __m128i _mm_maskz_shufflehi_epi16( __mmask8 k, __m128i a, int n );
(V)PSHUFHW __m128i _mm_shufflehi_epi16(__m128i a, int n)
VPSHUFHW __m256i _mm256_shufflehi_epi16(__m256i a, const int n)
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-52, “Type E4NF Class Exception Conditions.”

Additionally:

#UD	If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.

PSHUFLW—Shuffle Packed Low Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 70 /r ib PSHUFLW xmm1, xmm2/m128, imm8	A	V/V	SSE2	Shuffle the low words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.128.F2.0F.WIG 70 /r ib VPSHUFLW xmm1, xmm2/m128, imm8	A	V/V	AVX	Shuffle the low words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.256.F2.0F.WIG 70 /r ib VPSHUFLW ymm1, ymm2/m256, imm8	A	V/V	AVX2	Shuffle the low words in ymm2/m256 based on the encoding in imm8 and store the result in ymm1.
EVEX.128.F2.0F.WIG 70 /r ib VPSHUFLW xmm1 {k1}{z}, xmm2/m128, imm8	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shuffle the low words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1 under write mask k1.
EVEX.256.F2.0F.WIG 70 /r ib VPSHUFLW ymm1 {k1}{z}, ymm2/m256, imm8	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shuffle the low words in ymm2/m256 based on the encoding in imm8 and store the result in ymm1 under write mask k1.
EVEX.512.F2.0F.WIG 70 /r ib VPSHUFLW zmm1 {k1}{z}, zmm2/m512, imm8	B	V/V	AVX512BW OR AVX10.1	Shuffle the low words in zmm2/m512 based on the encoding in imm8 and store the result in zmm1 under write mask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
B	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Copies words from the low quadword of a 128-bit lane of the source operand and inserts them in the low quadword of the destination operand at word locations (of the respective lane) selected with the immediate operand. The 256-bit operation is similar to the in-lane operation used by the 256-bit VPSHUFD instruction, which is illustrated in Figure 4-16. For 128-bit operation, only the low 128-bit lane is operative. Each 2-bit field in the immediate operand selects the contents of one word location in the low quadword of the destination operand. The binary encodings of the immediate operand fields select words (0, 1, 2 or 3) from the low quadword of the source operand to be copied to the destination operand. The high quadword of the source operand is copied to the high quadword of the destination operand, for each 128-bit lane.

Note that this instruction permits a word in the low quadword of the source operand to be copied to more than one word location in the low quadword of the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The destination operand is an YMM register. The source operand can be an YMM register or a 256-bit memory location.

EVEX encoded version: The destination operand is a ZMM/YMM/XMM registers. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the write-mask.

Note: In VEX encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

PSHUFLW (128-bit Legacy SSE Version)

```
DEST[15:0] := (SRC >> (imm[1:0] *16))[15:0]
DEST[31:16] := (SRC >> (imm[3:2] * 16))[15:0]
DEST[47:32] := (SRC >> (imm[5:4] * 16))[15:0]
DEST[63:48] := (SRC >> (imm[7:6] * 16))[15:0]
DEST[127:64] := SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

VPSHUFLW (VEX.128 Encoded Version)

```
DEST[15:0] := (SRC1 >> (imm[1:0] *16))[15:0]
DEST[31:16] := (SRC1 >> (imm[3:2] * 16))[15:0]
DEST[47:32] := (SRC1 >> (imm[5:4] * 16))[15:0]
DEST[63:48] := (SRC1 >> (imm[7:6] * 16))[15:0]
DEST[127:64] := SRC[127:64]
DEST[MAXVL-1:128] := 0
```

VPSHUFLW (VEX.256 Encoded Version)

```
DEST[15:0] := (SRC1 >> (imm[1:0] *16))[15:0]
DEST[31:16] := (SRC1 >> (imm[3:2] * 16))[15:0]
DEST[47:32] := (SRC1 >> (imm[5:4] * 16))[15:0]
DEST[63:48] := (SRC1 >> (imm[7:6] * 16))[15:0]
DEST[127:64] := SRC1[127:64]
DEST[143:128] := (SRC1 >> (imm[1:0] *16))[143:128]
DEST[159:144] := (SRC1 >> (imm[3:2] * 16))[143:128]
DEST[175:160] := (SRC1 >> (imm[5:4] * 16))[143:128]
DEST[191:176] := (SRC1 >> (imm[7:6] * 16))[143:128]
DEST[255:192] := SRC1[255:192]
DEST[MAXVL-1:256] := 0
```

VPSHUFLW (EVEX.U1.512 Encoded Version)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL >= 128
    TMP_DEST[15:0] := (SRC1 >> (imm[1:0] *16))[15:0]
    TMP_DEST[31:16] := (SRC1 >> (imm[3:2] * 16))[15:0]
    TMP_DEST[47:32] := (SRC1 >> (imm[5:4] * 16))[15:0]
    TMP_DEST[63:48] := (SRC1 >> (imm[7:6] * 16))[15:0]
    TMP_DEST[127:64] := SRC1[127:64]
FI;
IF VL >= 256
    TMP_DEST[143:128] := (SRC1 >> (imm[1:0] *16))[143:128]
    TMP_DEST[159:144] := (SRC1 >> (imm[3:2] * 16))[143:128]
    TMP_DEST[175:160] := (SRC1 >> (imm[5:4] * 16))[143:128]
    TMP_DEST[191:176] := (SRC1 >> (imm[7:6] * 16))[143:128]
    TMP_DEST[255:192] := SRC1[255:192]
FI;
IF VL >= 512
    TMP_DEST[271:256] := (SRC1 >> (imm[1:0] *16))[271:256]
    TMP_DEST[287:272] := (SRC1 >> (imm[3:2] * 16))[271:256]
    TMP_DEST[303:288] := (SRC1 >> (imm[5:4] * 16))[271:256]
    TMP_DEST[319:304] := (SRC1 >> (imm[7:6] * 16))[271:256]
    TMP_DEST[383:320] := SRC1[383:320]
```

```
    TMP_DEST[399:384] := (SRC1 >> (imm[1:0] *16))[399:384]
    TMP_DEST[415:400] := (SRC1 >> (imm[3:2] * 16))[399:384]
    TMP_DEST[431:416] := (SRC1 >> (imm[5:4] * 16))[399:384]
    TMP_DEST[447:432] := (SRC1 >> (imm[7:6] * 16))[399:384]
    TMP_DEST[511:448] := SRC1[511:448]
FI;

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TMP_DEST[i+15:i];
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFLW __m512i _mm512_shufflelo_epi16(__m512i a, int n);
VPSHUFLW __m512i _mm512_mask_shufflelo_epi16(__m512i s, __mmask16 k, __m512i a, int n);
VPSHUFLW __m512i _mm512_maskz_shufflelo_epi16(__mmask16 k, __m512i a, int n);
VPSHUFLW __m256i _mm256_mask_shufflelo_epi16(__m256i s, __mmask8 k, __m256i a, int n);
VPSHUFLW __m256i _mm256_maskz_shufflelo_epi16(__mmask8 k, __m256i a, int n);
VPSHUFLW __m128i _mm_mask_shufflelo_epi16(__m128i s, __mmask8 k, __m128i a, int n);
VPSHUFLW __m128i _mm_maskz_shufflelo_epi16(__mmask8 k, __m128i a, int n);
(V)PSHUFLW:__m128i _mm_shufflelo_epi16(__m128i a, int n)
VPSHUFLW:__m256i _mm256_shufflelo_epi16(__m256i a, const int n)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."
Additionally:

#UD	If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.

PSHUFW—Shuffle Packed Words

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F 70 /r ib PSHUFW mm1, mm2/m64, imm8	RMI	Valid	Valid	Shuffle the words in mm2/m64 based on the encoding in imm8 and store the result in mm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Copies words from the source operand (second operand) and inserts them in the destination operand (first operand) at word locations selected with the order operand (third operand). This operation is similar to the operation used by the PSHUFD instruction, which is illustrated in Figure 4-16. For the PSHUFW instruction, each 2-bit field in the order operand selects the contents of one word location in the destination operand. The encodings of the order operand fields select words from the source operand to be copied to the destination operand.

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register. The order operand is an 8-bit immediate. Note that this instruction permits a word in the source operand to be copied to more than one word location in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Operation

```
DEST[15:0] := (SRC >> (ORDER[1:0] * 16))[15:0];
DEST[31:16] := (SRC >> (ORDER[3:2] * 16))[15:0];
DEST[47:32] := (SRC >> (ORDER[5:4] * 16))[15:0];
DEST[63:48] := (SRC >> (ORDER[7:6] * 16))[15:0];
```

Intel C/C++ Compiler Intrinsic Equivalent

PSHUFW __m64 _mm_shuffle_pi16(__m64 a, int n)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

PSIGNB/PSIGNW/PSIGND—Packed SIGN

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 08 /r[1] PSIGNB mm1, mm2/m64	RM	V/V	SSSE3	Negate/zero/preserve packed byte integers in mm1 depending on the corresponding sign in mm2/m64.
66 0F 38 08 /r PSIGNB xmm1, xmm2/m128	RM	V/V	SSSE3	Negate/zero/preserve packed byte integers in xmm1 depending on the corresponding sign in xmm2/m128.
NP 0F 38 09 /r[1] PSIGNW mm1, mm2/m64	RM	V/V	SSSE3	Negate/zero/preserve packed word integers in mm1 depending on the corresponding sign in mm2/m128.
66 0F 38 09 /r PSIGNW xmm1, xmm2/m128	RM	V/V	SSSE3	Negate/zero/preserve packed word integers in xmm1 depending on the corresponding sign in xmm2/m128.
NP 0F 38 0A /r[1] PSIGND mm1, mm2/m64	RM	V/V	SSSE3	Negate/zero/preserve packed doubleword integers in mm1 depending on the corresponding sign in mm2/m128.
66 0F 38 0A /r PSIGND xmm1, xmm2/m128	RM	V/V	SSSE3	Negate/zero/preserve packed doubleword integers in xmm1 depending on the corresponding sign in xmm2/m128.
VEX.128.66.0F38.WIG 08 /r VPSIGNB xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Negate/zero/preserve packed byte integers in xmm2 depending on the corresponding sign in xmm3/m128.
VEX.128.66.0F38.WIG 09 /r VPSIGNW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Negate/zero/preserve packed word integers in xmm2 depending on the corresponding sign in xmm3/m128.
VEX.128.66.0F38.WIG 0A /r VPSIGND xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Negate/zero/preserve packed doubleword integers in xmm2 depending on the corresponding sign in xmm3/m128.
VEX.256.66.0F38.WIG 08 /r VPSIGNB ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Negate packed byte integers in ymm2 if the corresponding sign in ymm3/m256 is less than zero.
VEX.256.66.0F38.WIG 09 /r VPSIGNW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Negate packed 16-bit integers in ymm2 if the corresponding sign in ymm3/m256 is less than zero.
VEX.256.66.0F38.WIG 0A /r VPSIGND ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Negate packed doubleword integers in ymm2 if the corresponding sign in ymm3/m256 is less than zero.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

(V)PSIGNB/(V)PSIGNW/(V)PSIGND negates each data element of the destination operand (the first operand) if the signed integer value of the corresponding data element in the source operand (the second operand) is less than zero. If the signed integer value of a data element in the source operand is positive, the corresponding data element in the destination operand is unchanged. If a data element in the source operand is zero, the corresponding data element in the destination operand is set to zero.

(V)PSIGNB operates on signed bytes. (V)PSIGNW operates on 16-bit signed words. (V)PSIGND operates on signed 32-bit integers.

Legacy SSE instructions: Both operands can be MMX registers. In 64-bit mode, use the REX prefix to access additional registers.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand is an YMM register or a 256-bit memory location.

Operation

```
def byte_sign(control, input_val):
  if control<0:
    return negate(input_val)
  elif control==0:
    return 0
  return input_val

def word_sign(control, input_val):
  if control<0:
    return negate(input_val)
  elif control==0:
    return 0
  return input_val

def dword_sign(control, input_val):
  if control<0:
    return negate(input_val)
  elif control==0:
    return 0
  return input_val
```

PSIGNB srcdest, src // MMX 64-bit Operands

```
VL=64
KL := VL/8
for i in 0...KL-1:
  srcdest.byte[i] := byte_sign(src.byte[i], srcdest.byte[i])
```

PSIGNW srcdest, src // MMX 64-bit Operands

```
VL=64
KL := VL/16
FOR i in 0...KL-1:
  srcdest.word[i] := word_sign(src.word[i], srcdest.word[i])
```

PSIGND srcdest, src // MMX 64-bit Operands

```
VL=64
KL := VL/32
FOR i in 0...KL-1:
  srcdest.dword[i] := dword_sign(src.dword[i], srcdest.dword[i])
```

PSIGNB srcdest, src // SSE 128-bit Operands

```
VL=128
KL := VL/8
FOR i in 0...KL-1:
  srcdest.byte[i] := byte_sign(src.byte[i], srcdest.byte[i])
```

PSIGNW srcdest, src // SSE 128-bit Operands

```
VL=128
KL := VL/16
FOR i in 0...KL-1:
  srcdest.word[i] := word_sign(src.word[i], srcdest.word[i])
```

PSIGND srcdest, src // SSE 128-bit Operands

```
VL=128
KL := VL/32
FOR i in 0...KL-1:
  srcdest.dword[i] := dword_sign(src.dword[i], srcdest.dword[i])
```

VPSIGNB dest, src1, src2 // AVX 128-bit or 256-bit Operands

```
VL=(128,256)
KL := VL/8
FOR i in 0...KL-1:
  dest.byte[i] := byte_sign(src2.byte[i], src1.byte[i])
DEST[MAXVL-1:VL] := 0
```

VPSIGNW dest, src1, src2 // AVX 128-bit or 256-bit Operands

```
VL=(128,256)
KL := VL/16
FOR i in 0...KL-1:
  dest.word[i] := word_sign(src2.word[i], src1.word[i])
DEST[MAXVL-1:VL] := 0
```

VPSIGND dest, src1, src2 // AVX 128-bit or 256-bit Operands

```
VL=(128,256)
KL := VL/32
FOR i in 0...KL-1:
  dest.dword[i] := dword_sign(src2.dword[i], src1.dword[i])
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PSIGNB __m64 _mm_sign_pi8 (__m64 a, __m64 b)
(V)PSIGNB __m128i _mm_sign_epi8 (__m128i a, __m128i b)
VPSIGNB __m256i _mm256_sign_epi8 (__m256i a, __m256i b)
PSIGNW __m64 _mm_sign_pi16 (__m64 a, __m64 b)
(V)PSIGNW __m128i _mm_sign_epi16 (__m128i a, __m128i b)
VPSIGNW __m256i _mm256_sign_epi16 (__m256i a, __m256i b)
PSIGND __m64 _mm_sign_pi32 (__m64 a, __m64 b)
(V)PSIGND __m128i _mm_sign_epi32 (__m128i a, __m128i b)
VPSIGND __m256i _mm256_sign_epi32 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions," additionally:

#UD	If VEX.L = 1.

PSLLDQ—Shift Double Quadword Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 73 /7 ib PSLLDQ xmm1, imm8	A	V/V	SSE2	Shift xmm1 left by imm8 bytes while shifting in 0s.
VEX.128.66.0F.WIG 73 /7 ib VPSLLDQ xmm1, xmm2, imm8	B	V/V	AVX	Shift xmm2 left by imm8 bytes while shifting in 0s and store result in xmm1.
VEX.256.66.0F.WIG 73 /7 ib VPSLLDQ ymm1, ymm2, imm8	B	V/V	AVX2	Shift ymm2 left by imm8 bytes while shifting in 0s and store result in ymm1.
EVEX.128.66.0F.WIG 73 /7 ib VPSLLDQ xmm1,xmm2/ m128, imm8	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift xmm2/m128 left by imm8 bytes while shifting in 0s and store result in xmm1.
EVEX.256.66.0F.WIG 73 /7 ib VPSLLDQ ymm1, ymm2/m256, imm8	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift ymm2/m256 left by imm8 bytes while shifting in 0s and store result in ymm1.
EVEX.512.66.0F.WIG 73 /7 ib VPSLLDQ zmm1, zmm2/m512, imm8	C	V/V	AVX512BW OR AVX10.1	Shift zmm2/m512 left by imm8 bytes while shifting in 0s and store result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (r, w)	imm8	N/A	N/A
B	N/A	VEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
C	Full Mem	EVEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A

Description

Shifts the destination operand (first operand) to the left by the number of bytes specified in the count operand (second operand). The empty low-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s. The count operand is an 8-bit immediate.

128-bit Legacy SSE version: The source and destination operands are the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The source operand is YMM register. The destination operand is an YMM register. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed. The count operand applies to both the low and high 128-bit lanes.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register. The count operand applies to each 128-bit lanes.

Operation

VPSLLDQ (EVEX.U1.512 Encoded Version)

```
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST[127:0] := SRC[127:0] << (TEMP * 8)
DEST[255:128] := SRC[255:128] << (TEMP * 8)
DEST[383:256] := SRC[383:256] << (TEMP * 8)
DEST[511:384] := SRC[511:384] << (TEMP * 8)
DEST[MAXVL-1:512] := 0
```

VPSLLDQ (VEX.256 and EVEX.256 Encoded Version)

```
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST[127:0] := SRC[127:0] << (TEMP * 8)
DEST[255:128] := SRC[255:128] << (TEMP * 8)
DEST[MAXVL-1:256] := 0
```

VPSLLDQ (VEX.128 and EVEX.128 Encoded Version)

```
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST := SRC << (TEMP * 8)
DEST[MAXVL-1:128] := 0
```

PSLLDQ(128-bit Legacy SSE Version)

```
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST := DEST << (TEMP * 8)
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PSLLDQ __m128i _mm_slli_si128 (__m128i a, int imm)
VPSLLDQ __m256i _mm256_slli_si256 (__m256i a, const int imm)
VPSLLDQ __m512i _mm512_bslli_epi128 (__m512i a, const int imm)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-24, “Type 7 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-52, “Type E4NF Class Exception Conditions.”

PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F1 /r[1] PSLLW mm, mm/m64	A	V/V	MMX	Shift words in mm left mm/m64 while shifting in 0s.
66 0F F1 /r PSLLW xmm1, xmm2/m128	A	V/V	SSE2	Shift words in xmm1 left by xmm2/m128 while shifting in 0s.
NP 0F 71 /6 ib PSLLW mm1, imm8	B	V/V	MMX	Shift words in mm left by imm8 while shifting in 0s.
66 0F 71 /6 ib PSLLW xmm1, imm8	B	V/V	SSE2	Shift words in xmm1 left by imm8 while shifting in 0s.
NP 0F F2 /r[1] PSLLD mm, mm/m64	A	V/V	MMX	Shift doublewords in mm left by mm/m64 while shifting in 0s.
66 0F F2 /r PSLLD xmm1, xmm2/m128	A	V/V	SSE2	Shift doublewords in xmm1 left by xmm2/m128 while shifting in 0s.
NP 0F 72 /6 ib[1] PSLLD mm, imm8	B	V/V	MMX	Shift doublewords in mm left by imm8 while shifting in 0s.
66 0F 72 /6 ib PSLLD xmm1, imm8	B	V/V	SSE2	Shift doublewords in xmm1 left by imm8 while shifting in 0s.
NP 0F F3 /r[1] PSLLQ mm, mm/m64	A	V/V	MMX	Shift quadword in mm left by mm/m64 while shifting in 0s.
66 0F F3 /r PSLLQ xmm1, xmm2/m128	A	V/V	SSE2	Shift quadwords in xmm1 left by xmm2/m128 while shifting in 0s.
NP 0F 73 /6 ib[1] PSLLQ mm, imm8	B	V/V	MMX	Shift quadword in mm left by imm8 while shifting in 0s.
66 0F 73 /6 ib PSLLQ xmm1, imm8	B	V/V	SSE2	Shift quadwords in xmm1 left by imm8 while shifting in 0s.
VEX.128.66.0F.WIG F1 /r VPSLLW xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift words in xmm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.128.66.0F.WIG 71 /6 ib VPSLLW xmm1, xmm2, imm8	D	V/V	AVX	Shift words in xmm2 left by imm8 while shifting in 0s.
VEX.128.66.0F.WIG F2 /r VPSLLD xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift doublewords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.128.66.0F.WIG 72 /6 ib VPSLLD xmm1, xmm2, imm8	D	V/V	AVX	Shift doublewords in xmm2 left by imm8 while shifting in 0s.
VEX.128.66.0F.WIG F3 /r VPSLLQ xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift quadwords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.128.66.0F.WIG 73 /6 ib VPSLLQ xmm1, xmm2, imm8	D	V/V	AVX	Shift quadwords in xmm2 left by imm8 while shifting in 0s.
VEX.256.66.0F.WIG F1 /r VPSLLW ymm1, ymm2, xmm3/m128	C	V/V	AVX2	Shift words in ymm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.256.66.0F.WIG 71 /6 ib VPSLLW ymm1, ymm2, imm8	D	V/V	AVX2	Shift words in ymm2 left by imm8 while shifting in 0s.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F.WIG F2 /r VPSLLD ymm1, ymm2, xmm3/m128	C	V/V	AVX2	Shift doublewords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.256.66.0F.WIG 72 /6 ib VPSLLD ymm1, ymm2, imm8	D	V/V	AVX2	Shift doublewords in ymm2 left by imm8 while shifting in 0s.
VEX.256.66.0F.WIG F3 /r VPSLLQ ymm1, ymm2, xmm3/m128	C	V/V	AVX2	Shift quadwords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.256.66.0F.WIG 73 /6 ib VPSLLQ ymm1, ymm2, imm8	D	V/V	AVX2	Shift quadwords in ymm2 left by imm8 while shifting in 0s.
EVEX.128.66.0F.WIG F1 /r VPSLLW xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in xmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.256.66.0F.WIG F1 /r VPSLLW ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in ymm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.512.66.0F.WIG F1 /r VPSLLW zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512BW OR AVX10.1	Shift words in zmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.128.66.0F.WIG 71 /6 ib VPSLLW xmm1 {k1}{z}, xmm2/m128, imm8	E	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in xmm2/m128 left by imm8 while shifting in 0s using writemask k1.
EVEX.256.66.0F.WIG 71 /6 ib VPSLLW ymm1 {k1}{z}, ymm2/m256, imm8	E	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in ymm2/m256 left by imm8 while shifting in 0s using writemask k1.
EVEX.512.66.0F.WIG 71 /6 ib VPSLLW zmm1 {k1}{z}, zmm2/m512, imm8	E	V/V	AVX512BW OR AVX10.1	Shift words in zmm2/m512 left by imm8 while shifting in 0 using writemask k1.
EVEX.128.66.0F.W0 F2 /r VPSLLD xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s under writemask k1.
EVEX.256.66.0F.W0 F2 /r VPSLLD ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s under writemask k1.
EVEX.512.66.0F.W0 F2 /r VPSLLD zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F OR AVX10.1	Shift doublewords in zmm2 left by amount specified in xmm3/m128 while shifting in 0s under writemask k1.
EVEX.128.66.0F.W0 72 /6 ib VPSLLD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in xmm2/m128/m32bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.256.66.0F.W0 72 /6 ib VPSLLD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in ymm2/m256/m32bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W0 72 /6 ib VPSLLD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	F	V/V	AVX512F OR AVX10.1	Shift doublewords in zmm2/m512/m32bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.128.66.0F.W1 F3 /r VPSLLQ xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F.W1 F3 /r VPSLLQ ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W1 F3 /r VPSLLQ zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F OR AVX10.1	Shift quadwords in zmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.128.66.0F.W1 73 /6 ib VPSLLQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in xmm2/m128/m64bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.256.66.0F.W1 73 /6 ib VPSLLQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in ymm2/m256/m64bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W1 73 /6 ib VPSLLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	F	V/V	AVX512F OR AVX10.1	Shift quadwords in zmm2/m512/m64bcst left by imm8 while shifting in 0s using writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:r/m (r, w)	imm8	N/A	N/A
C	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
D	N/A	VEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
E	Full Mem	EVEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
F	Full	EVEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
G	Mem128	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the left by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-17 gives an example of shifting words in a 64-bit operand.

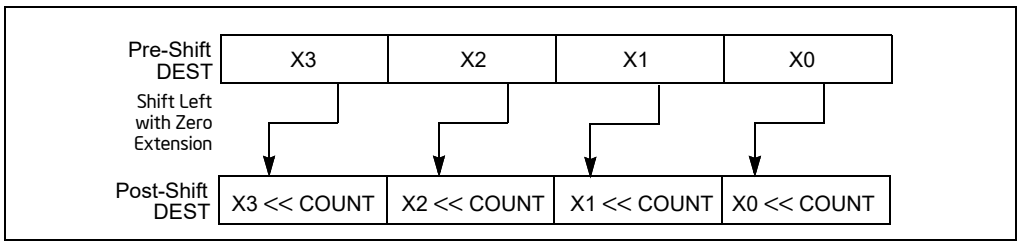


Figure 4-17. PSLLW, PSLLD, and PSLLQ Instruction Operation Using 64-bit Operand

The (V)PSLLW instruction shifts each of the words in the destination operand to the left by the number of bits specified in the count operand; the (V)PSLLD instruction shifts each of the doublewords in the destination operand; and the (V)PSLLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.

128-bit Legacy SSE version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /6, or EVEX.128.66.0F 71-73 /6), VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

PSLLW (With 64-bit Operand)

```
IF (COUNT > 15)
THEN
    DEST[64:0] := 0000000000000000H;
ELSE
    DEST[15:0] := ZeroExtend(DEST[15:0] << COUNT);
    (* Repeat shift operation for 2nd and 3rd words *)
    DEST[63:48] := ZeroExtend(DEST[63:48] << COUNT);
FI;
```

PSLLD (with 64-bit operand)

```
IF (COUNT > 31)
THEN
    DEST[64:0] := 0000000000000000H;
ELSE
    DEST[31:0] := ZeroExtend(DEST[31:0] << COUNT);
    DEST[63:32] := ZeroExtend(DEST[63:32] << COUNT);
FI;
```

PSLLQ (With 64-bit Operand)

```
IF (COUNT > 63)
THEN
    DEST[64:0] := 0000000000000000H;
ELSE
```

```
        DEST := ZeroExtend(DEST << COUNT);
    FI;

LOGICAL_LEFT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
ELSE
    DEST[15:0] := ZeroExtend(SRC[15:0] << COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] := ZeroExtend(SRC[127:112] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[31:0] := 0
ELSE
    DEST[31:0] := ZeroExtend(SRC[31:0] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
ELSE
    DEST[31:0] := ZeroExtend(SRC[31:0] << COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[127:96] := ZeroExtend(SRC[127:96] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[63:0] := 0
ELSE
    DEST[63:0] := ZeroExtend(SRC[63:0] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
ELSE
    DEST[63:0] := ZeroExtend(SRC[63:0] << COUNT);
    DEST[127:64] := ZeroExtend(SRC[127:64] << COUNT);
FI;
LOGICAL_LEFT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
```

```
IF (COUNT > 15)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
    DEST[255:128] := 00000000000000000000000000000000H
ELSE
    DEST[15:0] := ZeroExtend(SRC[15:0] << COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
    DEST[255:240] := ZeroExtend(SRC[255:240] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
    DEST[255:128] := 00000000000000000000000000000000H
ELSE
    DEST[31:0] := ZeroExtend(SRC[31:0] << COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[255:224] := ZeroExtend(SRC[255:224] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
    DEST[255:128] := 00000000000000000000000000000000H
ELSE
    DEST[63:0] := ZeroExtend(SRC[63:0] << COUNT);
    DEST[127:64] := ZeroExtend(SRC[127:64] << COUNT)
    DEST[191:128] := ZeroExtend(SRC[191:128] << COUNT);
    DEST[255:192] := ZeroExtend(SRC[255:192] << COUNT);
FI;
```

VPSLLW (EVEX Versions, xmm/m128)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] := LOGICAL_LEFT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] := LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] := LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] := LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
```

```
                    THEN *DEST[i+15:i] remains unchanged*
                    ELSE *zeroing-masking*                  ; zeroing-masking
                        DEST[i+15:i] = 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSLLW (EVEX Versions, imm8)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] := LOGICAL_LEFT_SHIFT_WORDS_128b(SRC1[127:0], imm8)
FI;
IF VL = 256
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
FI;
IF VL = 512
    TMP_DEST[255:0] := LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
    TMP_DEST[511:256] := LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[511:256], imm8)
FI;

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSLLW (ymm, ymm, xmm/m128) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0;
```

VPSLLW (ymm, imm8) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_LEFT_SHIFT_WORD_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0;
```

VPSLLW (xmm, xmm, xmm/m128) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPSLLW (xmm, imm8) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_WORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0
```

PSLLW (xmm, xmm, xmm/m128)

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_WORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSLLW (xmm, imm8)

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_WORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSLLD (EVEX versions, imm8)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+31:i] := LOGICAL_LEFT_SHIFT_DWORDS1(SRC1[31:0], imm8)
                ELSE DEST[i+31:i] := LOGICAL_LEFT_SHIFT_DWORDS1(SRC1[i+31:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSLLD (EVEX Versions, xmm/m128)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    TMP_DEST[127:0] := LOGICAL_LEFT_SHIFT_DWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] := LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] := LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] := LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSLLD (ymm, ymm, xmm/m128) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0;
```

VPSLLD (ymm, imm8) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0;
```

VPSLLD (xmm, xmm, xmm/m128) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPSLLD (xmm, imm8) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0
```

PSLLD (xmm, xmm, xmm/m128)

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSLLD (xmm, imm8)

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_DWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSLLQ (EVEX Versions, imm8)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+63:i] := LOGICAL_LEFT_SHIFT_QWORDS1(SRC1[63:0], imm8)
                ELSE DEST[i+63:i] := LOGICAL_LEFT_SHIFT_QWORDS1(SRC1[i+63:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
```

VPSLLQ (EVEX Versions, xmm/m128)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF VL = 128
    TMP_DEST[127:0] := LOGICAL_LEFT_SHIFT_QWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] := LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] := LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] := LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j := 0 TO KL-1
    i := j * 64
```

```
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSLLQ (ymm, ymm, xmm/m128) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0;
```

VPSLLQ (ymm, imm8) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0;
```

VPSLLQ (xmm, xmm, xmm/m128) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPSLLQ (xmm, imm8) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_QWORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0
```

PSLLQ (xmm, xmm, xmm/m128)

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSLLQ (xmm, imm8)

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_QWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPSLLD __m512i _mm512_slli_epi32(__m512i a, unsigned int imm);
VPSLLD __m512i _mm512_mask_slli_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSLLD __m512i _mm512_maskz_slli_epi32( __mmask16 k, __m512i a, unsigned int imm);
VPSLLD __m256i _mm256_mask_slli_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSLLD __m256i _mm256_maskz_slli_epi32( __mmask8 k, __m256i a, unsigned int imm);
VPSLLD __m128i _mm_mask_slli_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLD __m128i _mm_maskz_slli_epi32( __mmask8 k, __m128i a, unsigned int imm);
VPSLLD __m512i _mm512_sll_epi32(__m512i a, __m128i cnt);
VPSLLD __m512i _mm512_mask_sll_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSLLD __m512i _mm512_maskz_sll_epi32( __mmask16 k, __m512i a, __m128i cnt);
VPSLLD __m256i _mm256_mask_sll_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSLLD __m256i _mm256_maskz_sll_epi32( __mmask8 k, __m256i a, __m128i cnt);
VPSLLD __m128i _mm_mask_sll_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLD __m128i _mm_maskz_sll_epi32( __mmask8 k, __m128i a, __m128i cnt);
VPSLLQ __m512i _mm512_mask_slli_epi64(__m512i a, unsigned int imm);
VPSLLQ __m512i _mm512_mask_slli_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm);
```

VPSLLQ __m512i _mm512_maskz_slli_epi64(__mmask8 k, __m512i a, unsigned int imm);
VPSLLQ __m256i _mm256_mask_slli_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSLLQ __m256i _mm256_maskz_slli_epi64(__mmask8 k, __m256i a, unsigned int imm);
VPSLLQ __m128i _mm_mask_slli_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLQ __m128i _mm_maskz_slli_epi64(__mmask8 k, __m128i a, unsigned int imm);
VPSLLQ __m512i _mm512_mask_sll_epi64(__m512i a, __m128i cnt);
VPSLLQ __m512i _mm512_mask_sll_epi64(__m512i s, __mmask8 k, __m512i a, __m128i cnt);
VPSLLQ __m512i _mm512_maskz_sll_epi64(__mmask8 k, __m512i a, __m128i cnt);
VPSLLQ __m256i _mm256_mask_sll_epi64(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSLLQ __m256i _mm256_maskz_sll_epi64(__mmask8 k, __m256i a, __m128i cnt);
VPSLLQ __m128i _mm_mask_sll_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLQ __m128i _mm_maskz_sll_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSLLW __m512i _mm512_slli_epi16(__m512i a, unsigned int imm);
VPSLLW __m512i _mm512_mask_slli_epi16(__m512i s, __mmask32 k, __m512i a, unsigned int imm);
VPSLLW __m512i _mm512_maskz_slli_epi16(__mmask32 k, __m512i a, unsigned int imm);
VPSLLW __m256i _mm256_mask_slli_epi16(__m256i s, __mmask16 k, __m256i a, unsigned int imm);
VPSLLW __m256i _mm256_maskz_slli_epi16(__mmask16 k, __m256i a, unsigned int imm);
VPSLLW __m128i _mm_mask_slli_epi16(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLW __m128i _mm_maskz_slli_epi16(__mmask8 k, __m128i a, unsigned int imm);
VPSLLW __m512i _mm512_sll_epi16(__m512i a, __m128i cnt);
VPSLLW __m512i _mm512_mask_sll_epi16(__m512i s, __mmask32 k, __m512i a, __m128i cnt);
VPSLLW __m512i _mm512_maskz_sll_epi16(__mmask32 k, __m512i a, __m128i cnt);
VPSLLW __m256i _mm256_mask_sll_epi16(__m256i s, __mmask16 k, __m256i a, __m128i cnt);
VPSLLW __m256i _mm256_maskz_sll_epi16(__mmask16 k, __m256i a, __m128i cnt);
VPSLLW __m128i _mm_mask_sll_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLW __m128i _mm_maskz_sll_epi16(__mmask8 k, __m128i a, __m128i cnt);
PSLLW __m64 _mm_slli_pi16 (__m64 m, int count)
PSLLW __m64 _mm_sll_pi16(__m64 m, __m64 count)
(V)PSLLW __m128i _mm_slli_epi16(__m64 m, int count)
(V)PSLLW __m128i _mm_sll_epi16(__m128i m, __m128i count)
VPSLLW __m256i _mm256_slli_epi16 (__m256i m, int count)
VPSLLW __m256i _mm256_sll_epi16 (__m256i m, __m128i count)
PSLLD __m64 _mm_slli_pi32(__m64 m, int count)
PSLLD __m64 _mm_sll_pi32(__m64 m, __m64 count)
(V)PSLLD __m128i _mm_slli_epi32(__m128i m, int count)
(V)PSLLD __m128i _mm_sll_epi32(__m128i m, __m128i count)
VPSLLD __m256i _mm256_slli_epi32 (__m256i m, int count)
VPSLLD __m256i _mm256_sll_epi32 (__m256i m, __m128i count)
PSLLQ __m64 _mm_slli_si64(__m64 m, int count)
PSLLQ __m64 _mm_sll_si64(__m64 m, __m64 count)
(V)PSLLQ __m128i _mm_slli_epi64(__m128i m, int count)
(V)PSLLQ __m128i _mm_sll_epi64(__m128i m, __m128i count)
VPSLLQ __m256i _mm256_slli_epi64 (__m256i m, int count)
VPSLLQ __m256i _mm256_sll_epi64 (__m256i m, __m128i count)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

- VEX-encoded instructions:

  - Syntax with RM/RVM operand encoding (A/C in the operand encoding table), see Table 2-21, "Type 4 Class Exception Conditions."
  - Syntax with MI/VMI operand encoding (B/D in the operand encoding table), see Table 2-24, "Type 7 Class Exception Conditions."
- EVEX-encoded VPSLLW:
  - Syntax with Mem128 tuple type (G in the operand encoding table), see Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."
  - Syntax with Full tuple type (E in the operand encoding table), see Exceptions Type E4.nb in Table 2-51, "Type E4 Class Exception Conditions."
- EVEX-encoded VPSLLD/Q:
  - Syntax with Mem128 tuple type (G in the operand encoding table), see Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."
  - Syntax with Full tuple type (F in the operand encoding table), see Table 2-51, "Type E4 Class Exception Conditions."

PSRAW/PSRAD/PSRAQ—Shift Packed Data Right Arithmetic

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E1 /r[1] PSRAW mm, mm/m64	A	V/V	MMX	Shift words in mm right by mm/m64 while shifting in sign bits.
66 0F E1 /r PSRAW xmm1, xmm2/m128	A	V/V	SSE2	Shift words in xmm1 right by xmm2/m128 while shifting in sign bits.
NP 0F 71 /4 ib[1] PSRAW mm, imm8	B	V/V	MMX	Shift words in mm right by imm8 while shifting in sign bits
66 0F 71 /4 ib PSRAW xmm1, imm8	B	V/V	SSE2	Shift words in xmm1 right by imm8 while shifting in sign bits
NP 0F E2 /r[1] PSRAD mm, mm/m64	A	V/V	MMX	Shift doublewords in mm right by mm/m64 while shifting in sign bits.
66 0F E2 /r PSRAD xmm1, xmm2/m128	A	V/V	SSE2	Shift doubleword in xmm1 right by xmm2 /m128 while shifting in sign bits.
NP 0F 72 /4 ib[1] PSRAD mm, imm8	B	V/V	MMX	Shift doublewords in mm right by imm8 while shifting in sign bits.
66 0F 72 /4 ib PSRAD xmm1, imm8	B	V/V	SSE2	Shift doublewords in xmm1 right by imm8 while shifting in sign bits.
VEX.128.66.0F.WIG E1 /r VPSRAW xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits.
VEX.128.66.0F.WIG 71 /4 ib VPSRAW xmm1, xmm2, imm8	D	V/V	AVX	Shift words in xmm2 right by imm8 while shifting in sign bits.
VEX.128.66.0F.WIG E2 /r VPSRAD xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits.
VEX.128.66.0F.WIG 72 /4 ib VPSRAD xmm1, xmm2, imm8	D	V/V	AVX	Shift doublewords in xmm2 right by imm8 while shifting in sign bits.
VEX.256.66.0F.WIG E1 /r VPSRAW ymm1, ymm2, xmm3/m128	C	V/V	AVX2	Shift words in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits.
VEX.256.66.0F.WIG 71 /4 ib VPSRAW ymm1, ymm2, imm8	D	V/V	AVX2	Shift words in ymm2 right by imm8 while shifting in sign bits.
VEX.256.66.0F.WIG E2 /r VPSRAD ymm1, ymm2, xmm3/m128	C	V/V	AVX2	Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits.
VEX.256.66.0F.WIG 72 /4 ib VPSRAD ymm1, ymm2, imm8	D	V/V	AVX2	Shift doublewords in ymm2 right by imm8 while shifting in sign bits.
EVEX.128.66.0F.WIG E1 /r VPSRAW xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.256.66.0F.WIG E1 /r VPSRAW ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.512.66.0F.WIG E1 /r VPSRAW zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512BW OR AVX10.1	Shift words in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.WIG 71 /4 ib VPSRAW xmm1 {k1}{z}, xmm2/m128, imm8	E	V/V	(AVX512VL AND AVX512BW)OR AVX10.1[2]	Shift words in xmm2/m128 right by imm8 while shifting in sign bits using writemask k1.
EVEX.256.66.0F.WIG 71 /4 ib VPSRAW ymm1 {k1}{z}, ymm2/m256, imm8	E	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in ymm2/m256 right by imm8 while shifting in sign bits using writemask k1.
EVEX.512.66.0F.WIG 71 /4 ib VPSRAW zmm1 {k1}{z}, zmm2/m512, imm8	E	V/V	AVX512BW OR AVX10.1	Shift words in zmm2/m512 right by imm8 while shifting in sign bits using writemask k1.
EVEX.128.66.0F.W0 E2 /r VPSRAD xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.256.66.0F.W0 E2 /r VPSRAD ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.512.66.0F.W0 E2 /r VPSRAD zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F OR AVX10.1	Shift doublewords in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.128.66.0F.W0 72 /4 ib VPSRAD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in xmm2/m128/m32bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.256.66.0F.W0 72 /4 ib VPSRAD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in ymm2/m256/m32bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.512.66.0F.W0 72 /4 ib VPSRAD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	F	V/V	AVX512F OR AVX10.1	Shift doublewords in zmm2/m512/m32bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.128.66.0F.W1 E2 /r VPSRAQ xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.256.66.0F.W1 E2 /r VPSRAQ ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.512.66.0F.W1 E2 /r VPSRAQ zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F OR AVX10.1	Shift quadwords in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.128.66.0F.W1 72 /4 ib VPSRAQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in xmm2/m128/m64bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.256.66.0F.W1 72 /4 ib VPSRAQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in ymm2/m256/m64bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.512.66.0F.W1 72 /4 ib VPSRAQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	F	V/V	AVX512F OR AVX10.1	Shift quadwords in zmm2/m512/m64bcst right by imm8 while shifting in sign bits using writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:r/m (r, w)	imm8	N/A	N/A
C	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
D	N/A	VEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
E	Full Mem	EVEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
F	Full	EVEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
G	Mem128	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Shifts the bits in the individual data elements (words, doublewords or quadwords) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are filled with the initial value of the sign bit of the data element. If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for quadwords), each destination data element is filled with the initial value of the sign bit of the element. (Figure 4-18 gives an example of shifting words in a 64-bit operand.)

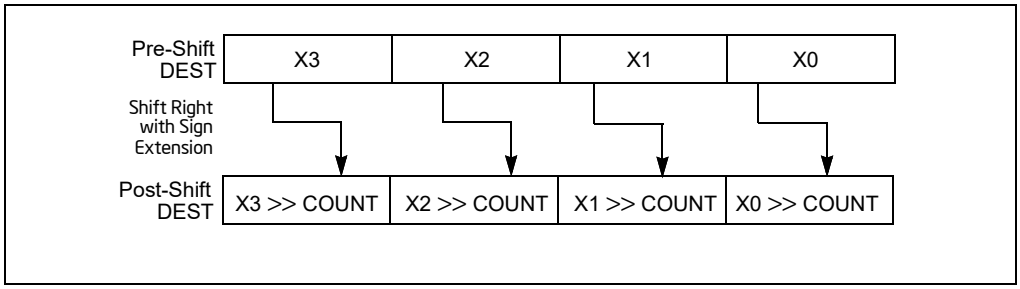


Figure 4-18. PSRAW and PSRAD Instruction Operation Using a 64-bit Operand

Note that only the first 64-bits of a 128-bit count operand are checked to compute the count. If the second source operand is a memory address, 128 bits are loaded.

The (V)PSRAW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand, and the (V)PSRAD instruction shifts each of the doublewords in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.

128-bit Legacy SSE version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /4, EVEX.128.66.0F 71-73 /4), VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

PSRAW (With 64-bit Operand)

```
    IF (COUNT > 15)
        THEN COUNT := 16;
    FI;
    DEST[15:0] := SignExtend(DEST[15:0] >> COUNT);
    (* Repeat shift operation for 2nd and 3rd words *)
    DEST[63:48] := SignExtend(DEST[63:48] >> COUNT);
```

```
PSRAD (with 64-bit operand)
    IF (COUNT > 31)
        THEN COUNT := 32;
    FI;
    DEST[31:0] := SignExtend(DEST[31:0] >> COUNT);
    DEST[63:32] := SignExtend(DEST[63:32] >> COUNT);
```

```
ARITHMETIC_RIGHT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[31:0] := SignBit
ELSE
    DEST[31:0] := SignExtend(SRC[31:0] >> COUNT);
FI;
```

```
ARITHMETIC_RIGHT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[63:0] := SignBit
ELSE
    DEST[63:0] := SignExtend(SRC[63:0] >> COUNT);
FI;
```

```
ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 15)
    THEN     COUNT := 16;
FI;
DEST[15:0] := SignExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
DEST[255:240] := SignExtend(SRC[255:240] >> COUNT);
```

```
ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
    THEN    COUNT := 32;
FI;
DEST[31:0] := SignExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
DEST[255:224] := SignExtend(SRC[255:224] >> COUNT);

ARITHMETIC_RIGHT_SHIFT_QWORDS(SRC, COUNT_SRC, VL) ; VL: 128b, 256b or 512b
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
    THEN    COUNT := 64;
FI;
DEST[63:0] := SignExtend(SRC[63:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
DEST[VL-1:VL-64] := SignExtend(SRC[VL-1:VL-64] >> COUNT);

ARITHMETIC_RIGHT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 15)
    THEN    COUNT := 16;
FI;
DEST[15:0] := SignExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
DEST[127:112] := SignExtend(SRC[127:112] >> COUNT);

ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
    THEN    COUNT := 32;
FI;
DEST[31:0] := SignExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
DEST[127:96] := SignExtend(SRC[127:96] >> COUNT);
```

VPSRAW (EVEX versions, xmm/m128)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TMP_DEST[i+15:i]
```

```
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRAW (EVEX Versions, imm8)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], imm8)
FI;
IF VL = 256
    TMP_DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
FI;
IF VL = 512
    TMP_DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
    TMP_DEST[511:256] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], imm8)
FI;

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRAW (ymm, ymm, xmm/m128) - VEX

```
DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPSRAW (ymm, imm8) - VEX

```
DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0
```

VPSRAW (xmm, xmm, xmm/m128) - VEX

```
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPSRAW (xmm, imm8) - VEX

```
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0
```

PSRAW (xmm, xmm, xmm/m128)

```
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRAW (xmm, imm8)

```
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSRAD (EVEX Versions, imm8)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+31:i] := ARITHMETIC_RIGHT_SHIFT_DWORDS1(SRC1[31:0], imm8)
                ELSE DEST[i+31:i] := ARITHMETIC_RIGHT_SHIFT_DWORDS1(SRC1[i+31:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRAD (EVEX Versions, xmm/m128)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    TMP_DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] := ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRAD (ymm, ymm, xmm/m128) - VEX

```
DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPSRAD (ymm, imm8) - VEX

```
DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0
```

VPSRAD (xmm, xmm, xmm/m128) - VEX

```
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPSRAD (xmm, imm8) - VEX

```
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0
```

PSRAD (xmm, xmm, xmm/m128)

```
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRAD (xmm, imm8)

```
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSRAQ (EVEX Versions, imm8)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+63:i] := ARITHMETIC_RIGHT_SHIFT_QWORDS1(SRC1[63:0], imm8)
                ELSE DEST[i+63:i] := ARITHMETIC_RIGHT_SHIFT_QWORDS1(SRC1[i+63:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRAQ (EVEX Versions, xmm/m128)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
TMP_DEST[VL-1:0] := ARITHMETIC_RIGHT_SHIFT_QWORDS(SRC1[VL-1:0], SRC2, VL)

FOR j := 0 TO 7
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
```

```
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPSRAD __m512i _mm512_srai_epi32(__m512i a, unsigned int imm);
VPSRAD __m512i _mm512_mask_srai_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSRAD __m512i _mm512_maskz_srai_epi32(__mmask16 k, __m512i a, unsigned int imm);
VPSRAD __m256i _mm256_mask_srai_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRAD __m256i _mm256_maskz_srai_epi32(__mmask8 k, __m256i a, unsigned int imm);
VPSRAD __m128i _mm_mask_srai_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRAD __m128i _mm_maskz_srai_epi32(__mmask8 k, __m128i a, unsigned int imm);
VPSRAD __m512i _mm512_sra_epi32(__m512i a, __m128i cnt);
VPSRAD __m512i _mm512_mask_sra_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSRAD __m512i _mm512_maskz_sra_epi32(__mmask16 k, __m512i a, __m128i cnt);
VPSRAD __m256i _mm256_mask_sra_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRAD __m256i _mm256_maskz_sra_epi32(__mmask8 k, __m256i a, __m128i cnt);
VPSRAD __m128i _mm_mask_sra_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRAD __m128i _mm_maskz_sra_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPSRAQ __m512i _mm512_srai_epi64(__m512i a, unsigned int imm);
VPSRAQ __m512i _mm512_mask_srai_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm)
VPSRAQ __m512i _mm512_maskz_srai_epi64(__mmask8 k, __m512i a, unsigned int imm)
VPSRAQ __m256i _mm256_mask_srai_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRAQ __m256i _mm256_maskz_srai_epi64(__mmask8 k, __m256i a, unsigned int imm);
VPSRAQ __m128i _mm_mask_srai_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRAQ __m128i _mm_maskz_srai_epi64(__mmask8 k, __m128i a, unsigned int imm);
VPSRAQ __m512i _mm512_sra_epi64(__m512i a, __m128i cnt);
VPSRAQ __m512i _mm512_mask_sra_epi64(__m512i s, __mmask8 k, __m512i a, __m128i cnt)
VPSRAQ __m512i _mm512_maskz_sra_epi64(__mmask8 k, __m512i a, __m128i cnt)
VPSRAQ __m256i _mm256_mask_sra_epi64(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRAQ __m256i _mm256_maskz_sra_epi64(__mmask8 k, __m256i a, __m128i cnt);
VPSRAQ __m128i _mm_mask_sra_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRAQ __m128i _mm_maskz_sra_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSRAW __m512i _mm512_srai_epi16(__m512i a, unsigned int imm);
VPSRAW __m512i _mm512_mask_srai_epi16(__m512i s, __mmask32 k, __m512i a, unsigned int imm);
VPSRAW __m512i _mm512_maskz_srai_epi16(__mmask32 k, __m512i a, unsigned int imm);
VPSRAW __m256i _mm256_mask_srai_epi16(__m256i s, __mmask16 k, __m256i a, unsigned int imm);
VPSRAW __m256i _mm256_maskz_srai_epi16(__mmask16 k, __m256i a, unsigned int imm);
VPSRAW __m128i _mm_mask_srai_epi16(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRAW __m128i _mm_maskz_srai_epi16(__mmask8 k, __m128i a, unsigned int imm);
VPSRAW __m512i _mm512_sra_epi16(__m512i a, __m128i cnt);
VPSRAW __m512i _mm512_mask_sra_epi16(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSRAW __m512i _mm512_maskz_sra_epi16(__mmask16 k, __m512i a, __m128i cnt);
VPSRAW __m256i _mm256_mask_sra_epi16(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRAW __m256i _mm256_maskz_sra_epi16(__mmask8 k, __m256i a, __m128i cnt);
VPSRAW __m128i _mm_mask_sra_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRAW __m128i _mm_maskz_sra_epi16(__mmask8 k, __m128i a, __m128i cnt);
PSRAW __m64 _mm_srai_pi16 (__m64 m, int count)
PSRAW __m64 _mm_sra_pi16 (__m64 m, __m64 count)
(V)PSRAW __m128i _mm_srai_epi16(__m128i m, int count)
(V)PSRAW __m128i _mm_sra_epi16(__m128i m, __m128i count)

VPSRAW __m256i _mm256_srai_epi16 (__m256i m, int count)
VPSRAW __m256i _mm256_sra_epi16 (__m256i m, __m128i count)
PSRAD __m64 _mm_srai_pi32 (__m64 m, int count)
PSRAD __m64 _mm_sra_pi32 (__m64 m, __m64 count)
(V)PSRAD __m128i _mm_srai_epi32 (__m128i m, int count)
(V)PSRAD __m128i _mm_sra_epi32 (__m128i m, __m128i count)
VPSRAD __m256i _mm256_srai_epi32 (__m256i m, int count)
VPSRAD __m256i _mm256_sra_epi32 (__m256i m, __m128i count)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

- VEX-encoded instructions:
  - Syntax with RM/RVM operand encoding (A/C in the operand encoding table), see Table 2-21, "Type 4 Class Exception Conditions."
  - Syntax with MI/VMI operand encoding (B/D in the operand encoding table), see Table 2-24, "Type 7 Class Exception Conditions."
- EVEX-encoded VPSRAW:
  - Syntax with Mem128 tuple type (G in the operand encoding table), see Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."
  - Syntax with Full tuple type (E in the operand encoding table), see Exceptions Type E4.nb in Table 2-51, "Type E4 Class Exception Conditions."
- EVEX-encoded VPSRAD/Q:
  - Syntax with Mem128 tuple type (G in the operand encoding table), see Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."
  - Syntax with Full tuple type (F in the operand encoding table), see Table 2-51, "Type E4 Class Exception Conditions."

PSRLDQ—Shift Double Quadword Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 73 /3 ib PSRLDQ xmm1, imm8	A	V/V	SSE2	Shift xmm1 right by imm8 while shifting in 0s.
VEX.128.66.0F.WIG 73 /3 ib VPSRLDQ xmm1, xmm2, imm8	B	V/V	AVX	Shift xmm2 right by imm8 bytes while shifting in 0s.
VEX.256.66.0F.WIG 73 /3 ib VPSRLDQ ymm1, ymm2, imm8	B	V/V	AVX2	Shift ymm1 right by imm8 bytes while shifting in 0s.
EVEX.128.66.0F.WIG 73 /3 ib VPSRLDQ xmm1, xmm2/m128, imm8	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift xmm2/m128 right by imm8 bytes while shifting in 0s and store result in xmm1.
EVEX.256.66.0F.WIG 73 /3 ib VPSRLDQ ymm1, ymm2/m256, imm8	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift ymm2/m256 right by imm8 bytes while shifting in 0s and store result in ymm1.
EVEX.512.66.0F.WIG 73 /3 ib VPSRLDQ zmm1, zmm2/m512, imm8	C	V/V	AVX512BW OR AVX10.1	Shift zmm2/m512 right by imm8 bytes while shifting in 0s and store result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (r, w)	imm8	N/A	N/A
B	N/A	VEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
C	Full Mem	EVEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A

Description

Shifts the destination operand (first operand) to the right by the number of bytes specified in the count operand (second operand). The empty high-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s. The count operand is an 8-bit immediate.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The source and destination operands are the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is an YMM register. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed. The count operand applies to both the low and high 128-bit lanes.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register. The count operand applies to each 128-bit lanes.

Note: VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

VPSRLDQ (EVEX.512 Encoded Version)

```
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST[127:0] := SRC[127:0] >> (TEMP * 8)
DEST[255:128] := SRC[255:128] >> (TEMP * 8)
DEST[383:256] := SRC[383:256] >> (TEMP * 8)
DEST[511:384] := SRC[511:384] >> (TEMP * 8)
DEST[MAXVL-1:512] := 0;
```

VPSRLDQ (VEX.256 and EVEX.256 Encoded Version)

```
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST[127:0] := SRC[127:0] >> (TEMP * 8)
DEST[255:128] := SRC[255:128] >> (TEMP * 8)
DEST[MAXVL-1:256] := 0;
```

VPSRLDQ (VEX.128 and EVEX.128 Encoded Version)

```
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST := SRC >> (TEMP * 8)
DEST[MAXVL-1:128] := 0;
```

PSRLDQ (128-bit Legacy SSE Version)

```
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST := DEST >> (TEMP * 8)
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalents

(V)PSRLDQ __m128i _mm_srli_si128 (__m128i a, int imm)
VPSRLDQ __m256i _mm256_bsrli_epi128 (__m256i, const int)
VPSRLDQ __m512i _mm512_bsrli_epi128 (__m512i, int)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-24, “Type 7 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-52, “Type E4NF Class Exception Conditions.”

PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F D1 /r[1] PSRLW mm, mm/m64	A	V/V	MMX	Shift words in mm right by amount specified in mm/m64 while shifting in 0s.
66 0F D1 /r PSRLW xmm1, xmm2/m128	A	V/V	SSE2	Shift words in xmm1 right by amount specified in xmm2/m128 while shifting in 0s.
NP 0F 71 /2 ib[1] PSRLW mm, imm8	B	V/V	MMX	Shift words in mm right by imm8 while shifting in 0s.
66 0F 71 /2 ib PSRLW xmm1, imm8	B	V/V	SSE2	Shift words in xmm1 right by imm8 while shifting in 0s.
NP 0F D2 /r[1] PSRLD mm, mm/m64	A	V/V	MMX	Shift doublewords in mm right by amount specified in mm/m64 while shifting in 0s.
66 0F D2 /r PSRLD xmm1, xmm2/m128	A	V/V	SSE2	Shift doublewords in xmm1 right by amount specified in xmm2 /m128 while shifting in 0s.
NP 0F 72 /2 ib[1] PSRLD mm, imm8	B	V/V	MMX	Shift doublewords in mm right by imm8 while shifting in 0s.
66 0F 72 /2 ib PSRLD xmm1, imm8	B	V/V	SSE2	Shift doublewords in xmm1 right by imm8 while shifting in 0s.
NP 0F D3 /r[1] PSRLQ mm, mm/m64	A	V/V	MMX	Shift mm right by amount specified in mm/m64 while shifting in 0s.
66 0F D3 /r PSRLQ xmm1, xmm2/m128	A	V/V	SSE2	Shift quadwords in xmm1 right by amount specified in xmm2/m128 while shifting in 0s.
NP 0F 73 /2 ib[1] PSRLQ mm, imm8	B	V/V	MMX	Shift mm right by imm8 while shifting in 0s.
66 0F 73 /2 ib PSRLQ xmm1, imm8	B	V/V	SSE2	Shift quadwords in xmm1 right by imm8 while shifting in 0s.
VEX.128.66.0F.WIG D1 /r VPSRLW xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.128.66.0F.WIG 71 /2 ib VPSRLW xmm1, xmm2, imm8	D	V/V	AVX	Shift words in xmm2 right by imm8 while shifting in 0s.
VEX.128.66.0F.WIG D2 /r VPSRLD xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.128.66.0F.WIG 72 /2 ib VPSRLD xmm1, xmm2, imm8	D	V/V	AVX	Shift doublewords in xmm2 right by imm8 while shifting in 0s.
VEX.128.66.0F.WIG D3 /r VPSRLQ xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.128.66.0F.WIG 73 /2 ib VPSRLQ xmm1, xmm2, imm8	D	V/V	AVX	Shift quadwords in xmm2 right by imm8 while shifting in 0s.
VEX.256.66.0F.WIG D1 /r VPSRLW ymm1, ymm2, xmm3/m128	C	V/V	AVX2	Shift words in ymm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.256.66.0F.WIG 71 /2 ib VPSRLW ymm1, ymm2, imm8	D	V/V	AVX2	Shift words in ymm2 right by imm8 while shifting in 0s.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F.WIG D2 /r VPSRLD ymm1, ymm2, xmm3/m128	C	V/V	AVX2	Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.256.66.0F.WIG 72 /2 ib VPSRLD ymm1, ymm2, imm8	D	V/V	AVX2	Shift doublewords in ymm2 right by imm8 while shifting in 0s.
VEX.256.66.0F.WIG D3 /r VPSRLQ ymm1, ymm2, xmm3/m128	C	V/V	AVX2	Shift quadwords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.256.66.0F.WIG 73 /2 ib VPSRLQ ymm1, ymm2, imm8	D	V/V	AVX2	Shift quadwords in ymm2 right by imm8 while shifting in 0s.
EVEX.128.66.0F.WIG D1 /r VPSRLW xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.256.66.0F.WIG D1 /r VPSRLW ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.512.66.0F.WIG D1 /r VPSRLW zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512BW OR AVX10.1	Shift words in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.128.66.0F.WIG 71 /2 ib VPSRLW xmm1 {k1}{z}, xmm2/m128, imm8	E	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in xmm2/m128 right by imm8 while shifting in 0s using writemask k1.
EVEX.256.66.0F.WIG 71 /2 ib VPSRLW ymm1 {k1}{z}, ymm2/m256, imm8	E	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in ymm2/m256 right by imm8 while shifting in 0s using writemask k1.
EVEX.512.66.0F.WIG 71 /2 ib VPSRLW zmm1 {k1}{z}, zmm2/m512, imm8	E	V/V	AVX512BW OR AVX10.1	Shift words in zmm2/m512 right by imm8 while shifting in 0s using writemask k1.
EVEX.128.66.0F.W0 D2 /r VPSRLD xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.256.66.0F.W0 D2 /r VPSRLD ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W0 D2 /r VPSRLD zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F OR AVX10.1	Shift doublewords in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.128.66.0F.W0 72 /2 ib VPSRLD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in xmm2/m128/m32bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.256.66.0F.W0 72 /2 ib VPSRLD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in ymm2/m256/m32bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W0 72 /2 ib VPSRLD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	F	V/V	AVX512F OR AVX10.1	Shift doublewords in zmm2/m512/m32bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.128.66.0F.W1 D3 /r VPSRLQ xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F.W1 D3 /r VPSRLQ ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W1 D3 /r VPSRLQ zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F OR AVX10.1	Shift quadwords in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.128.66.0F.W1 73 /2 ib VPSRLQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in xmm2/m128/m64bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.256.66.0F.W1 73 /2 ib VPSRLQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in ymm2/m256/m64bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W1 73 /2 ib VPSRLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	F	V/V	AVX512F OR AVX10.1	Shift quadwords in zmm2/m512/m64bcst right by imm8 while shifting in 0s using writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:r/m (r, w)	imm8	N/A	N/A
C	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
D	N/A	VEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
E	Full Mem	EVEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
F	Full	EVEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
G	Mem128	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-19 gives an example of shifting words in a 64-bit operand.

Note that only the low 64-bits of a 128-bit count operand are checked to compute the count.

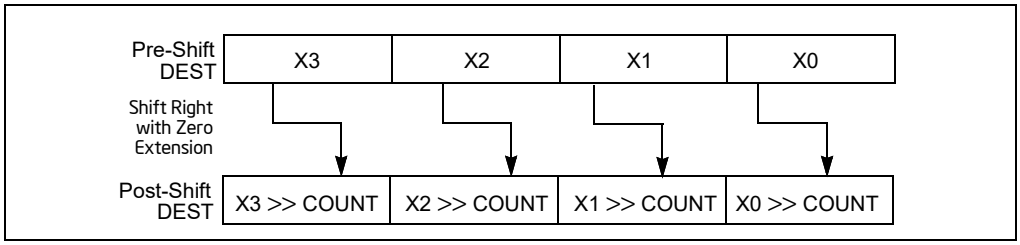


Figure 4-19. PSRLW, PSRLD, and PSRLQ Instruction Operation Using 64-bit Operand

The (V)PSRLW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand; the (V)PSRLD instruction shifts each of the doublewords in the destination operand; and the PSRLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instruction 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.

128-bit Legacy SSE version: The destination operand is an XMM register; the count operand can be either an XMM register or a 128-bit memory location, or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register; the count operand can be either an XMM register or a 128-bit memory location, or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /2, or EVEX.128.66.0F 71-73 /2), VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

PSRLW (With 64-bit Operand)

```
IF (COUNT > 15)
THEN
    DEST[64:0] := 0000000000000000H
ELSE
    DEST[15:0] := ZeroExtend(DEST[15:0] >> COUNT);
    (* Repeat shift operation for 2nd and 3rd words *)
    DEST[63:48] := ZeroExtend(DEST[63:48] >> COUNT);
FI;
```

PSRLD (With 64-bit Operand)

```
    IF (COUNT > 31)
    THEN
        DEST[64:0] := 0000000000000000H
    ELSE
        DEST[31:0] := ZeroExtend(DEST[31:0] >> COUNT);
        DEST[63:32] := ZeroExtend(DEST[63:32] >> COUNT);
    FI;
```

PSRLQ (With 64-bit Operand)

```
    IF (COUNT > 63)
    THEN
        DEST[64:0] := 0000000000000000H
    ELSE
        DEST := ZeroExtend(DEST >> COUNT);
    FI;
LOGICAL_RIGHT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[31:0] := 0
ELSE
    DEST[31:0] := ZeroExtend(SRC[31:0] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[63:0] := 0
ELSE
    DEST[63:0] := ZeroExtend(SRC[63:0] >> COUNT);
FI;
LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[255:0] := 0
ELSE
    DEST[15:0] := ZeroExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
    DEST[255:240] := ZeroExtend(SRC[255:240] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
ELSE
    DEST[15:0] := ZeroExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] := ZeroExtend(SRC[127:112] >> COUNT);
FI;
```

```
LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[255:0] := 0
ELSE
    DEST[31:0] := ZeroExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[255:224] := ZeroExtend(SRC[255:224] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
ELSE
    DEST[31:0] := ZeroExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[127:96] := ZeroExtend(SRC[127:96] >> COUNT);
FI;
LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[255:0] := 0
ELSE
    DEST[63:0] := ZeroExtend(SRC[63:0] >> COUNT);
    DEST[127:64] := ZeroExtend(SRC[127:64] >> COUNT);
    DEST[191:128] := ZeroExtend(SRC[191:128] >> COUNT);
    DEST[255:192] := ZeroExtend(SRC[255:192] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
ELSE
    DEST[63:0] := ZeroExtend(SRC[63:0] >> COUNT);
    DEST[127:64] := ZeroExtend(SRC[127:64] >> COUNT);
FI;
```

VPSRLW (EVEX Versions, xmm/m128)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] := LOGICAL_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
```

```
    TMP_DEST[511:256] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRLW (EVEX Versions, imm8)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] := LOGICAL_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], imm8)
FI;
IF VL = 256
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
FI;
IF VL = 512
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
    TMP_DEST[511:256] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], imm8)
FI;

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRLW (ymm, ymm, xmm/m128) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0;
```

VPSRLW (ymm, imm8) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0;
```

VPSRLW (xmm, xmm, xmm/m128) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPSRLW (xmm, imm8) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0
```

PSRLW (xmm, xmm, xmm/m128)

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRLW (xmm, imm8)

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSRLD (EVEX Versions, xmm/m128)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    TMP_DEST[127:0] := LOGICAL_RIGHT_SHIFT_DWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] := LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRLD (EVEX Versions, imm8)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+31:i] := LOGICAL_RIGHT_SHIFT_DWORDS1(SRC1[31:0], imm8)
                ELSE DEST[i+31:i] := LOGICAL_RIGHT_SHIFT_DWORDS1(SRC1[i+31:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
```

```
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE *zeroing-masking*                  ; zeroing-masking
                        DEST[i+31:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRLD (ymm, ymm, xmm/m128) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0;
```

VPSRLD (ymm, imm8) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0;
```

VPSRLD (xmm, xmm, xmm/m128) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPSRLD (xmm, imm8) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0
```

PSRLD (xmm, xmm, xmm/m128)

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRLD (xmm, imm8)

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSRLQ (EVEX Versions, xmm/m128)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
TMP_DEST[511:256] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)
IF VL = 128
    TMP_DEST[127:0] := LOGICAL_RIGHT_SHIFT_QWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
```

```
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRLQ (EVEX Versions, imm8)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+63:i] := LOGICAL_RIGHT_SHIFT_QWORDS1(SRC1[63:0], imm8)
                ELSE DEST[i+63:i] := LOGICAL_RIGHT_SHIFT_QWORDS1(SRC1[i+63:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRLQ (ymm, ymm, xmm/m128) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0;
```

VPSRLQ (ymm, imm8) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0;
```

VPSRLQ (xmm, xmm, xmm/m128) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPSRLQ (xmm, imm8) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0
```

PSRLQ (xmm, xmm, xmm/m128)

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRLQ (xmm, imm8)

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_QWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalents

VPSRLD __m512i _mm512_srli_epi32(__m512i a, unsigned int imm);
VPSRLD __m512i _mm512_mask_srli_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSRLD __m512i _mm512_maskz_srli_epi32(__mmask16 k, __m512i a, unsigned int imm);
VPSRLD __m256i _mm256_mask_srli_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);

VPSRLD __m256i _mm256_maskz_srli_epi32(__mmask8 k, __m256i a, unsigned int imm);
VPSRLD __m128i _mm_mask_srli_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLD __m128i _mm_maskz_srli_epi32(__mmask8 k, __m128i a, unsigned int imm);
VPSRLD __m512i _mm512_srl_epi32(__m512i a, __m128i cnt);
VPSRLD __m512i _mm512_mask_srl_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSRLD __m512i _mm512_maskz_srl_epi32(__mmask16 k, __m512i a, __m128i cnt);
VPSRLD __m256i _mm256_mask_srl_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRLD __m256i _mm256_maskz_srl_epi32(__mmask8 k, __m256i a, __m128i cnt);
VPSRLD __m128i _mm_mask_srl_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLD __m128i _mm_maskz_srl_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPSRLQ __m512i _mm512_srli_epi64(__m512i a, unsigned int imm);
VPSRLQ __m512i _mm512_mask_srli_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm);
VPSRLQ __m512i _mm512_mask_srli_epi64(__mmask8 k, __m512i a, unsigned int imm);
VPSRLQ __m256i _mm256_mask_srli_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRLQ __m256i _mm256_maskz_srli_epi64(__mmask8 k, __m256i a, unsigned int imm);
VPSRLQ __m128i _mm_mask_srli_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLQ __m128i _mm_maskz_srli_epi64(__mmask8 k, __m128i a, unsigned int imm);
VPSRLQ __m512i _mm512_srl_epi64(__m512i a, __m128i cnt);
VPSRLQ __m512i _mm512_mask_srl_epi64(__m512i s, __mmask8 k, __m512i a, __m128i cnt);
VPSRLQ __m512i _mm512_mask_srl_epi64(__mmask8 k, __m512i a, __m128i cnt);
VPSRLQ __m256i _mm256_mask_srl_epi64(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRLQ __m256i _mm256_maskz_srl_epi64(__mmask8 k, __m256i a, __m128i cnt);
VPSRLQ __m128i _mm_mask_srl_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLQ __m128i _mm_maskz_srl_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSRLW __m512i _mm512_srli_epi16(__m512i a, unsigned int imm);
VPSRLW __m512i _mm512_mask_srli_epi16(__m512i s, __mmask32 k, __m512i a, unsigned int imm);
VPSRLW __m512i _mm512_maskz_srli_epi16(__mmask32 k, __m512i a, unsigned int imm);
VPSRLW __m256i _mm256_mask_srli_epi16(__m256i s, __mmask16 k, __m256i a, unsigned int imm);
VPSRLW __m256i _mm256_maskz_srli_epi16(__mmask16 k, __m256i a, unsigned int imm);
VPSRLW __m128i _mm_mask_srli_epi16(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLW __m128i _mm_maskz_srli_epi16(__mmask8 k, __m128i a, unsigned int imm);
VPSRLW __m512i _mm512_srl_epi16(__m512i a, __m128i cnt);
VPSRLW __m512i _mm512_mask_srl_epi16(__m512i s, __mmask32 k, __m512i a, __m128i cnt);
VPSRLW __m512i _mm512_maskz_srl_epi16(__mmask32 k, __m512i a, __m128i cnt);
VPSRLW __m256i _mm256_mask_srl_epi16(__m256i s, __mmask16 k, __m256i a, __m128i cnt);
VPSRLW __m256i _mm256_maskz_srl_epi16(__mmask8 k, __mmask16 a, __m128i cnt);
VPSRLW __m128i _mm_mask_srl_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLW __m128i _mm_maskz_srl_epi16(__mmask8 k, __m128i a, __m128i cnt);
PSRLW __m64 _mm_srli_pi16(__m64 m, int count)
PSRLW __m64 _mm_srl_pi16 (__m64 m, __m64 count)
(V)PSRLW __m128i _mm_srli_epi16 (__m128i m, int count)
(V)PSRLW __m128i _mm_srl_epi16 (__m128i m, __m128i count)
VPSRLW __m256i _mm256_srli_epi16 (__m256i m, int count)
VPSRLW __m256i _mm256_srl_epi16 (__m256i m, __m128i count)
PSRLD __m64 _mm_srli_pi32 (__m64 m, int count)
PSRLD __m64 _mm_srl_pi32 (__m64 m, __m64 count)
(V)PSRLD __m128i _mm_srli_epi32 (__m128i m, int count)
(V)PSRLD __m128i _mm_srl_epi32 (__m128i m, __m128i count)
VPSRLD __m256i _mm256_srli_epi32 (__m256i m, int count)
VPSRLD __m256i _mm256_srl_epi32 (__m256i m, __m128i count)
PSRLQ __m64 _mm_srli_si64 (__m64 m, int count)
PSRLQ __m64 _mm_srl_si64 (__m64 m, __m64 count)
(V)PSRLQ __m128i _mm_srli_epi64 (__m128i m, int count)
(V)PSRLQ __m128i _mm_srl_epi64 (__m128i m, __m128i count)

VPSRLQ __m256i _mm256_srli_epi64 (__m256i m, int count)
VPSRLQ __m256i _mm256_srl_epi64 (__m256i m, __m128i count)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

- VEX-encoded instructions:
  - Syntax with RM/RVM operand encoding (A/C in the operand encoding table), see Table 2-21, "Type 4 Class Exception Conditions."
  - Syntax with MI/VMI operand encoding (B/D in the operand encoding table), see Table 2-24, "Type 7 Class Exception Conditions."
- EVEX-encoded VPSRLW:
  - Syntax with Mem128 tuple type (G in the operand encoding table), see Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."
  - Syntax with Full tuple type (E in the operand encoding table), see Exceptions Type E4.nb in Table 2-51, "Type E4 Class Exception Conditions."
- EVEX-encoded VPSRLD/Q:
  - Syntax with Mem128 tuple type (G in the operand encoding table), see Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."
  - Syntax with Full tuple type (F in the operand encoding table), see Table 2-51, "Type E4 Class Exception Conditions."

PSUBB/PSUBW/PSUBD—Subtract Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F8 /r[1] PSUBB mm, mm/m64	A	V/V	MMX	Subtract packed byte integers in mm/m64 from packed byte integers in mm.
66 0F F8 /r PSUBB xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed byte integers in xmm2/m128 from packed byte integers in xmm1.
NP 0F F9 /r[1] PSUBW mm, mm/m64	A	V/V	MMX	Subtract packed word integers in mm/m64 from packed word integers in mm.
66 0F F9 /r PSUBW xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed word integers in xmm2/m128 from packed word integers in xmm1.
NP 0F FA /r[1] PSUBD mm, mm/m64	A	V/V	MMX	Subtract packed doubleword integers in mm/m64 from packed doubleword integers in mm.
66 0F FA /r PSUBD xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed doubleword integers in xmm2/mem128 from packed doubleword integers in xmm1.
VEX.128.66.0F.WIG F8 /r VPSUBB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed byte integers in xmm3/m128 from xmm2.
VEX.128.66.0F.WIG F9 /r VPSUBW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed word integers in xmm3/m128 from xmm2.
VEX.128.66.0F.WIG FA /r VPSUBD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed doubleword integers in xmm3/m128 from xmm2.
VEX.256.66.0F.WIG F8 /r VPSUBB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed byte integers in ymm3/m256 from ymm2.
VEX.256.66.0F.WIG F9 /r VPSUBW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed word integers in ymm3/m256 from ymm2.
VEX.256.66.0F.WIG FA /r VPSUBD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed doubleword integers in ymm3/m256 from ymm2.
EVEX.128.66.0F.WIG F8 /r VPSUBB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Subtract packed byte integers in xmm3/m128 from xmm2 and store in xmm1 using writemask k1.
EVEX.256.66.0F.WIG F8 /r VPSUBB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Subtract packed byte integers in ymm3/m256 from ymm2 and store in ymm1 using writemask k1.
EVEX.512.66.0F.WIG F8 /r VPSUBB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Subtract packed byte integers in zmm3/m512 from zmm2 and store in zmm1 using writemask k1.
EVEX.128.66.0F.WIG F9 /r VPSUBW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Subtract packed word integers in xmm3/m128 from xmm2 and store in xmm1 using writemask k1.
EVEX.256.66.0F.WIG F9 /r VPSUBW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Subtract packed word integers in ymm3/m256 from ymm2 and store in ymm1 using writemask k1.
EVEX.512.66.0F.WIG F9 /r VPSUBW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Subtract packed word integers in zmm3/m512 from zmm2 and store in zmm1 using writemask k1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 FA /r VPSUBD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Subtract packed doubleword integers in xmm3/m128/m32bcst from xmm2 and store in xmm1 using writemask k1.
EVEX.256.66.0F.W0 FA /r VPSUBD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Subtract packed doubleword integers in ymm3/m256/m32bcst from ymm2 and store in ymm1 using writemask k1.
EVEX.512.66.0F.W0 FA /r VPSUBD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F OR AVX10.1	Subtract packed doubleword integers in zmm3/m512/m32bcst from zmm2 and store in zmm1 using writemask k1

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD subtract of the packed integers of the source operand (second operand) from the packed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

The (V)PSUBB instruction subtracts packed byte integers. When an individual result is too large or too small to be represented in a byte, the result is wrapped around and the low 8 bits are written to the destination element.

The (V)PSUBW instruction subtracts packed word integers. When an individual result is too large or too small to be represented in a word, the result is wrapped around and the low 16 bits are written to the destination element.

The (V)PSUBD instruction subtracts packed doubleword integers. When an individual result is too large or too small to be represented in a doubleword, the result is wrapped around and the low 32 bits are written to the destination element.

Note that the (V)PSUBB, (V)PSUBW, and (V)PSUBD instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values upon which it operates.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSUBD: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPSUBB/W: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PSUBB (With 64-bit Operands)

```
DEST[7:0] := DEST[7:0] – SRC[7:0];
(* Repeat subtract operation for 2nd through 7th byte *)
DEST[63:56] := DEST[63:56] – SRC[63:56];
```

PSUBW (With 64-bit Operands)

```
DEST[15:0] := DEST[15:0] – SRC[15:0];
(* Repeat subtract operation for 2nd and 3rd word *)
DEST[63:48] := DEST[63:48] – SRC[63:48];
```

PSUBD (With 64-bit Operands)

```
DEST[31:0] := DEST[31:0] – SRC[31:0];
DEST[63:32] := DEST[63:32] – SRC[63:32];
```

PSUBD (With 128-bit Operands)

```
DEST[31:0] := DEST[31:0] – SRC[31:0];
(* Repeat subtract operation for 2nd and 3rd doubleword *)
DEST[127:96] := DEST[127:96] – SRC[127:96];
```

VPSUBB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SRC1[i+7:i] - SRC2[i+7:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPSUBW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SRC1[i+15:i] - SRC2[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPSUBD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := SRC1[i+31:i] - SRC2[31:0]
                ELSE DEST[i+31:i] := SRC1[i+31:i] - SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPSUBB (VEX.256 Encoded Version)

```
DEST[7:0] := SRC1[7:0]-SRC2[7:0]
DEST[15:8] := SRC1[15:8]-SRC2[15:8]
DEST[23:16] := SRC1[23:16]-SRC2[23:16]
DEST[31:24] := SRC1[31:24]-SRC2[31:24]
DEST[39:32] := SRC1[39:32]-SRC2[39:32]
DEST[47:40] := SRC1[47:40]-SRC2[47:40]
DEST[55:48] := SRC1[55:48]-SRC2[55:48]
DEST[63:56] := SRC1[63:56]-SRC2[63:56]
DEST[71:64] := SRC1[71:64]-SRC2[71:64]
DEST[79:72] := SRC1[79:72]-SRC2[79:72]
DEST[87:80] := SRC1[87:80]-SRC2[87:80]
DEST[95:88] := SRC1[95:88]-SRC2[95:88]
DEST[103:96] := SRC1[103:96]-SRC2[103:96]
DEST[111:104] := SRC1[111:104]-SRC2[111:104]
DEST[119:112] := SRC1[119:112]-SRC2[119:112]
DEST[127:120] := SRC1[127:120]-SRC2[127:120]
DEST[135:128] := SRC1[135:128]-SRC2[135:128]
DEST[143:136] := SRC1[143:136]-SRC2[143:136]
```

DEST[151:144] := SRC1[151:144]-SRC2[151:144]
DEST[159:152] := SRC1[159:152]-SRC2[159:152]
DEST[167:160] := SRC1[167:160]-SRC2[167:160]
DEST[175:168] := SRC1[175:168]-SRC2[175:168]
DEST[183:176] := SRC1[183:176]-SRC2[183:176]
DEST[191:184] := SRC1[191:184]-SRC2[191:184]
DEST[199:192] := SRC1[199:192]-SRC2[199:192]
DEST[207:200] := SRC1[207:200]-SRC2[207:200]
DEST[215:208] := SRC1[215:208]-SRC2[215:208]
DEST[223:216] := SRC1[223:216]-SRC2[223:216]
DEST[231:224] := SRC1[231:224]-SRC2[231:224]
DEST[239:232] := SRC1[239:232]-SRC2[239:232]
DEST[247:240] := SRC1[247:240]-SRC2[247:240]
DEST[255:248] := SRC1[255:248]-SRC2[255:248]
DEST[MAXVL-1:256] := 0

VPSUBB (VEX.128 Encoded Version)
DEST[7:0] := SRC1[7:0]-SRC2[7:0]
DEST[15:8] := SRC1[15:8]-SRC2[15:8]
DEST[23:16] := SRC1[23:16]-SRC2[23:16]
DEST[31:24] := SRC1[31:24]-SRC2[31:24]
DEST[39:32] := SRC1[39:32]-SRC2[39:32]
DEST[47:40] := SRC1[47:40]-SRC2[47:40]
DEST[55:48] := SRC1[55:48]-SRC2[55:48]
DEST[63:56] := SRC1[63:56]-SRC2[63:56]
DEST[71:64] := SRC1[71:64]-SRC2[71:64]
DEST[79:72] := SRC1[79:72]-SRC2[79:72]
DEST[87:80] := SRC1[87:80]-SRC2[87:80]
DEST[95:88] := SRC1[95:88]-SRC2[95:88]
DEST[103:96] := SRC1[103:96]-SRC2[103:96]
DEST[111:104] := SRC1[111:104]-SRC2[111:104]
DEST[119:112] := SRC1[119:112]-SRC2[119:112]
DEST[127:120] := SRC1[127:120]-SRC2[127:120]
DEST[MAXVL-1:128] := 0

PSUBB (128-bit Legacy SSE Version)
DEST[7:0] := DEST[7:0]-SRC[7:0]
DEST[15:8] := DEST[15:8]-SRC[15:8]
DEST[23:16] := DEST[23:16]-SRC[23:16]
DEST[31:24] := DEST[31:24]-SRC[31:24]
DEST[39:32] := DEST[39:32]-SRC[39:32]
DEST[47:40] := DEST[47:40]-SRC[47:40]
DEST[55:48] := DEST[55:48]-SRC[55:48]
DEST[63:56] := DEST[63:56]-SRC[63:56]
DEST[71:64] := DEST[71:64]-SRC[71:64]
DEST[79:72] := DEST[79:72]-SRC[79:72]
DEST[87:80] := DEST[87:80]-SRC[87:80]
DEST[95:88] := DEST[95:88]-SRC[95:88]
DEST[103:96] := DEST[103:96]-SRC[103:96]
DEST[111:104] := DEST[111:104]-SRC[111:104]
DEST[119:112] := DEST[119:112]-SRC[119:112]
DEST[127:120] := DEST[127:120]-SRC[127:120]
DEST[MAXVL-1:128] (Unmodified)

VPSUBW (VEX.256 Encoded Version)
DEST[15:0] := SRC1[15:0]-SRC2[15:0]
DEST[31:16] := SRC1[31:16]-SRC2[31:16]
DEST[47:32] := SRC1[47:32]-SRC2[47:32]
DEST[63:48] := SRC1[63:48]-SRC2[63:48]
DEST[79:64] := SRC1[79:64]-SRC2[79:64]
DEST[95:80] := SRC1[95:80]-SRC2[95:80]
DEST[111:96] := SRC1[111:96]-SRC2[111:96]
DEST[127:112] := SRC1[127:112]-SRC2[127:112]
DEST[143:128] := SRC1[143:128]-SRC2[143:128]
DEST[159:144] := SRC1[159:144]-SRC2[159:144]
DEST[175:160] := SRC1[175:160]-SRC2[175:160]
DEST[191:176] := SRC1[191:176]-SRC2[191:176]
DEST[207:192] := SRC1207:192]-SRC2[207:192]
DEST[223:208] := SRC1[223:208]-SRC2[223:208]
DEST[239:224] := SRC1[239:224]-SRC2[239:224]
DEST[255:240] := SRC1[255:240]-SRC2[255:240]
DEST[MAXVL-1:256] := 0

VPSUBW (VEX.128 Encoded Version)
DEST[15:0] := SRC1[15:0]-SRC2[15:0]
DEST[31:16] := SRC1[31:16]-SRC2[31:16]
DEST[47:32] := SRC1[47:32]-SRC2[47:32]
DEST[63:48] := SRC1[63:48]-SRC2[63:48]
DEST[79:64] := SRC1[79:64]-SRC2[79:64]
DEST[95:80] := SRC1[95:80]-SRC2[95:80]
DEST[111:96] := SRC1[111:96]-SRC2[111:96]
DEST[127:112] := SRC1[127:112]-SRC2[127:112]
DEST[MAXVL-1:128] := 0

PSUBW (128-bit Legacy SSE Version)
DEST[15:0] := DEST[15:0]-SRC[15:0]
DEST[31:16] := DEST[31:16]-SRC[31:16]
DEST[47:32] := DEST[47:32]-SRC[47:32]
DEST[63:48] := DEST[63:48]-SRC[63:48]
DEST[79:64] := DEST[79:64]-SRC[79:64]
DEST[95:80] := DEST[95:80]-SRC[95:80]
DEST[111:96] := DEST[111:96]-SRC[111:96]
DEST[127:112] := DEST[127:112]-SRC[127:112]
DEST[MAXVL-1:128] (Unmodified)

VPSUBD (VEX.256 Encoded Version)
DEST[31:0] := SRC1[31:0]-SRC2[31:0]
DEST[63:32] := SRC1[63:32]-SRC2[63:32]
DEST[95:64] := SRC1[95:64]-SRC2[95:64]
DEST[127:96] := SRC1[127:96]-SRC2[127:96]
DEST[159:128] := SRC1[159:128]-SRC2[159:128]
DEST[191:160] := SRC1[191:160]-SRC2[191:160]
DEST[223:192] := SRC1[223:192]-SRC2[223:192]
DEST[255:224] := SRC1[255:224]-SRC2[255:224]
DEST[MAXVL-1:256] := 0

VPSUBD (VEX.128 Encoded Version)
DEST[31:0] := SRC1[31:0]-SRC2[31:0]
DEST[63:32] := SRC1[63:32]-SRC2[63:32]
DEST[95:64] := SRC1[95:64]-SRC2[95:64]
DEST[127:96] := SRC1[127:96]-SRC2[127:96]
DEST[MAXVL-1:128] := 0

PSUBD (128-bit Legacy SSE Version)
DEST[31:0] := DEST[31:0]-SRC[31:0]
DEST[63:32] := DEST[63:32]-SRC[63:32]
DEST[95:64] := DEST[95:64]-SRC[95:64]
DEST[127:96] := DEST[127:96]-SRC[127:96]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalents

VPSUBB __m512i _mm512_sub_epi8(__m512i a, __m512i b);
VPSUBB __m512i _mm512_mask_sub_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSUBB __m512i _mm512_maskz_sub_epi8(__mmask64 k, __m512i a, __m512i b);
VPSUBB __m256i _mm256_mask_sub_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSUBB __m256i _mm256_maskz_sub_epi8(__mmask32 k, __m256i a, __m256i b);
VPSUBB __m128i _mm_mask_sub_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSUBB __m128i _mm_maskz_sub_epi8(__mmask16 k, __m128i a, __m128i b);
VPSUBW __m512i _mm512_sub_epi16(__m512i a, __m512i b);
VPSUBW __m512i _mm512_mask_sub_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPSUBW __m512i _mm512_maskz_sub_epi16(__mmask32 k, __m512i a, __m512i b);
VPSUBW __m256i _mm256_mask_sub_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPSUBW __m256i _mm256_maskz_sub_epi16(__mmask16 k, __m256i a, __m256i b);
VPSUBW __m128i _mm_mask_sub_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBW __m128i _mm_maskz_sub_epi16(__mmask8 k, __m128i a, __m128i b);
VPSUBD __m512i _mm512_sub_epi32(__m512i a, __m512i b);
VPSUBD __m512i _mm512_mask_sub_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPSUBD __m512i _mm512_maskz_sub_epi32(__mmask16 k, __m512i a, __m512i b);
VPSUBD __m256i _mm256_mask_sub_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPSUBD __m256i _mm256_maskz_sub_epi32(__mmask8 k, __m256i a, __m256i b);
VPSUBD __m128i _mm_mask_sub_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBD __m128i _mm_maskz_sub_epi32(__mmask8 k, __m128i a, __m128i b);
PSUBB __m64 _mm_sub_pi8(__m64 m1, __m64 m2)
(V)PSUBB __m128i _mm_sub_epi8 (__m128i a, __m128i b)
VPSUBB __m256i _mm256_sub_epi8 (__m256i a, __m256i b)
PSUBW __m64 _mm_sub_pi16(__m64 m1, __m64 m2)
(V)PSUBW __m128i _mm_sub_epi16 (__m128i a, __m128i b)
VPSUBW __m256i _mm256_sub_epi16 (__m256i a, __m256i b)
PSUBD __m64 _mm_sub_pi32(__m64 m1, __m64 m2)
(V)PSUBD __m128i _mm_sub_epi32 (__m128i a, __m128i b)
VPSUBD __m256i _mm256_sub_epi32 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded VPSUBD, see Table 2-51, “Type E4 Class Exception Conditions.”

EVEX-encoded VPSUBB/W, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

PSUBQ—Subtract Packed Quadword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F FB /r[1] PSUBQ mm1, mm2/m64	A	V/V	SSE2	Subtract quadword integer in mm1 from mm2 /m64.
66 0F FB /r PSUBQ xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed quadword integers in xmm1 from xmm2 /m128.
VEX.128.66.0F.WIG FB/r VPSUBQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed quadword integers in xmm3/m128 from xmm2.
VEX.256.66.0F.WIG FB /r VPSUBQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed quadword integers in ymm3/m256 from ymm2.
EVEX.128.66.0F.W1 FB /r VPSUBQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Subtract packed quadword integers in xmm3/m128/m64bcst from xmm2 and store in xmm1 using writemask k1.
EVEX.256.66.0F.W1 FB /r VPSUBQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Subtract packed quadword integers in ymm3/m256/m64bcst from ymm2 and store in ymm1 using writemask k1.
EVEX.512.66.0F.W1 FB/r VPSUBQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Subtract packed quadword integers in zmm3/m512/m64bcst from zmm2 and store in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. When packed quadword operands are used, a SIMD subtract is performed. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

Note that the (V)PSUBQ instruction can operate on either unsigned or signed (two's complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values upon which it operates.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be a quadword integer stored in an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSUBQ: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PSUBQ (With 64-Bit Operands)

```
DEST[63:0] := DEST[63:0] – SRC[63:0];
```

PSUBQ (With 128-Bit Operands)

```
DEST[63:0] := DEST[63:0] – SRC[63:0];
DEST[127:64] := DEST[127:64] – SRC[127:64];
```

VPSUBQ (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0]-SRC2[63:0]
DEST[127:64] := SRC1[127:64]-SRC2[127:64]
DEST[MAXVL-1:128] := 0
```

VPSUBQ (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0]-SRC2[63:0]
DEST[127:64] := SRC1[127:64]-SRC2[127:64]
DEST[191:128] := SRC1[191:128]-SRC2[191:128]
DEST[255:192] := SRC1[255:192]-SRC2[255:192]
DEST[MAXVL-1:256] := 0
```

VPSUBQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
                IF (EVEX.b = 1) AND (SRC2 *is memory*)
                     THEN DEST[i+63:i] := SRC1[i+63:i] - SRC2[63:0]
                     ELSE DEST[i+63:i] := SRC1[i+63:i] - SRC2[i+63:i]
                FI;
          ELSE
                IF *merging-masking*                    ; merging-masking
                     THEN *DEST[i+63:i] remains unchanged*
                     ELSE *zeroing-masking*                  ; zeroing-masking
                          DEST[i+63:i] := 0
                FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPSUBQ __m512i _mm512_sub_epi64(__m512i a, __m512i b);
VPSUBQ __m512i _mm512_mask_sub_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPSUBQ __m512i _mm512_maskz_sub_epi64(__mmask8 k, __m512i a, __m512i b);
VPSUBQ __m256i _mm256_mask_sub_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPSUBQ __m256i _mm256_maskz_sub_epi64(__mmask8 k, __m256i a, __m256i b);
VPSUBQ __m128i _mm_mask_sub_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBQ __m128i _mm_maskz_sub_epi64(__mmask8 k, __m128i a, __m128i b);
PSUBQ __m64 _mm_sub_si64(__m64 m1, __m64 m2)
(V)PSUBQ __m128i _mm_sub_epi64(__m128i m1, __m128i m2)
VPSUBQ __m256i _mm256_sub_epi64(__m256i m1, __m256i m2)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."

EVEX-encoded VPSUBQ, see Table 2-51, "Type E4 Class Exception Conditions."

PSUBSB/PSUBSW—Subtract Packed Signed Integers With Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E8 /r[1] PSUBSB mm, mm/m64	A	V/V	MMX	Subtract signed packed bytes in mm/m64 from signed packed bytes in mm and saturate results.
66 0F E8 /r PSUBSB xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed signed byte integers in xmm2/m128 from packed signed byte integers in xmm1 and saturate results.
NP 0F E9 /r[1] PSUBSW mm, mm/m64	A	V/V	MMX	Subtract signed packed words in mm/m64 from signed packed words in mm and saturate results.
66 0F E9 /r PSUBSW xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed signed word integers in xmm2/m128 from packed signed word integers in xmm1 and saturate results.
VEX.128.66.0F.WIG E8 /r VPSUBSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed signed byte integers in xmm3/m128 from packed signed byte integers in xmm2 and saturate results.
VEX.128.66.0F.WIG E9 /r VPSUBSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed signed word integers in xmm3/m128 from packed signed word integers in xmm2 and saturate results.
VEX.256.66.0F.WIG E8 /r VPSUBSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed signed byte integers in ymm3/m256 from packed signed byte integers in ymm2 and saturate results.
VEX.256.66.0F.WIG E9 /r VPSUBSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed signed word integers in ymm3/m256 from packed signed word integers in ymm2 and saturate results.
EVEX.128.66.0F.WIG E8 /r VPSUBSB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Subtract packed signed byte integers in xmm3/m128 from packed signed byte integers in xmm2 and saturate results and store in xmm1 using writemask k1.
EVEX.256.66.0F.WIG E8 /r VPSUBSB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Subtract packed signed byte integers in ymm3/m256 from packed signed byte integers in ymm2 and saturate results and store in ymm1 using writemask k1.
EVEX.512.66.0F.WIG E8 /r VPSUBSB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Subtract packed signed byte integers in zmm3/m512 from packed signed byte integers in zmm2 and saturate results and store in zmm1 using writemask k1.
EVEX.128.66.0F.WIG E9 /r VPSUBSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Subtract packed signed word integers in xmm3/m128 from packed signed word integers in xmm2 and saturate results and store in xmm1 using writemask k1.
EVEX.256.66.0F.WIG E9 /r VPSUBSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Subtract packed signed word integers in ymm3/m256 from packed signed word integers in ymm2 and saturate results and store in ymm1 using writemask k1.
EVEX.512.66.0F.WIG E9 /r VPSUBSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Subtract packed signed word integers in zmm3/m512 from packed signed word integers in zmm2 and saturate results and store in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD subtract of the packed signed integers of the source operand (second operand) from the packed signed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

The (V)PSUBSB instruction subtracts packed signed byte integers. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

The (V)PSUBSW instruction subtracts packed signed word integers. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PSUBSB (With 64-bit Operands)

```
DEST[7:0] := SaturateToSignedByte (DEST[7:0] – SRC (7:0]);
(* Repeat subtract operation for 2nd through 7th bytes *)
DEST[63:56] := SaturateToSignedByte (DEST[63:56] – SRC[63:56] );
```

PSUBSW (With 64-bit Operands)

```
    DEST[15:0] := SaturateToSignedWord (DEST[15:0] – SRC[15:0] );
    (* Repeat subtract operation for 2nd and 7th words *)
    DEST[63:48] := SaturateToSignedWord (DEST[63:48] – SRC[63:48] );
```

VPSUBSB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8;
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SaturateToSignedByte (SRC1[i+7:i] - SRC2[i+7:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] := 0;
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPSUBSW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SaturateToSignedWord (SRC1[i+15:i] - SRC2[i+15:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] := 0;
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

VPSUBSB (VEX.256 Encoded Version)

```
DEST[7:0] := SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 31th bytes *)
DEST[255:248] := SaturateToSignedByte (SRC1[255:248] - SRC2[255:248]);
DEST[MAXVL-1:256] := 0;
```

VPSUBSB (VEX.128 Encoded Version)

```
DEST[7:0] := SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] := SaturateToSignedByte (SRC1[127:120] - SRC2[127:120]);
DEST[MAXVL-1:128] := 0;
```

PSUBSB (128-bit Legacy SSE Version)

```
DEST[7:0] := SaturateToSignedByte (DEST[7:0] - SRC[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] := SaturateToSignedByte (DEST[127:120] - SRC[127:120]);
DEST[MAXVL-1:128] (Unmodified);
```

VPSUBSW (VEX.256 Encoded Version)
DEST[15:0] := SaturateToSignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 15th words *)
DEST[255:240] := SaturateToSignedWord (SRC1[255:240] - SRC2[255:240]);
DEST[MAXVL-1:256] := 0;

VPSUBSW (VEX.128 Encoded Version)
DEST[15:0] := SaturateToSignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] := SaturateToSignedWord (SRC1[127:112] - SRC2[127:112]);
DEST[MAXVL-1:128] := 0;

PSUBSW (128-bit Legacy SSE Version)
DEST[15:0] := SaturateToSignedWord (DEST[15:0] - SRC[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] := SaturateToSignedWord (DEST[127:112] - SRC[127:112]);
DEST[MAXVL-1:128] (Unmodified);

Intel C/C++ Compiler Intrinsic Equivalents

VPSUBSB __m512i _mm512_subs_epi8(__m512i a, __m512i b);
VPSUBSB __m512i _mm512_mask_subs_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSUBSB __m512i _mm512_maskz_subs_epi8(__mmask64 k, __m512i a, __m512i b);
VPSUBSB __m256i _mm256_mask_subs_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSUBSB __m256i _mm256_maskz_subs_epi8(__mmask32 k, __m256i a, __m256i b);
VPSUBSB __m128i _mm_mask_subs_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSUBSB __m128i _mm_maskz_subs_epi8(__mmask16 k, __m128i a, __m128i b);
VPSUBSW __m512i _mm512_subs_epi16(__m512i a, __m512i b);
VPSUBSW __m512i _mm512_mask_subs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPSUBSW __m512i _mm512_maskz_subs_epi16(__mmask32 k, __m512i a, __m512i b);
VPSUBSW __m256i _mm256_mask_subs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPSUBSW __m256i _mm256_maskz_subs_epi16(__mmask16 k, __m256i a, __m256i b);
VPSUBSW __m128i _mm_mask_subs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBSW __m128i _mm_maskz_subs_epi16(__mmask8 k, __m128i a, __m128i b);
PSUBSB __m64 _mm_subs_pi8(__m64 m1, __m64 m2)
(V)PSUBSB __m128i _mm_subs_epi8(__m128i m1, __m128i m2)
VPSUBSB __m256i _mm256_subs_epi8(__m256i m1, __m256i m2)
PSUBSW __m64 _mm_subs_pi16(__m64 m1, __m64 m2)
(V)PSUBSW __m128i _mm_subs_epi16(__m128i m1, __m128i m2)
VPSUBSW __m256i _mm256_subs_epi16(__m256i m1, __m256i m2)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, "Type E4 Class Exception Conditions."

PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers With Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F D8 /r[1] PSUBUSB mm, mm/m64	A	V/V	MMX	Subtract unsigned packed bytes in mm/m64 from unsigned packed bytes in mm and saturate result.
66 0F D8 /r PSUBUSB xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed unsigned byte integers in xmm2/m128 from packed unsigned byte integers in xmm1 and saturate result.
NP 0F D9 /r[1] PSUBUSW mm, mm/m64	A	V/V	MMX	Subtract unsigned packed words in mm/m64 from unsigned packed words in mm and saturate result.
66 0F D9 /r PSUBUSW xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed unsigned word integers in xmm2/m128 from packed unsigned word integers in xmm1 and saturate result.
VEX.128.66.0F.WIG D8 /r VPSUBUSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed unsigned byte integers in xmm3/m128 from packed unsigned byte integers in xmm2 and saturate result.
VEX.128.66.0F.WIG D9 /r VPSUBUSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed unsigned word integers in xmm3/m128 from packed unsigned word integers in xmm2 and saturate result.
VEX.256.66.0F.WIG D8 /r VPSUBUSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed unsigned byte integers in ymm3/m256 from packed unsigned byte integers in ymm2 and saturate result.
VEX.256.66.0F.WIG D9 /r VPSUBUSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed unsigned word integers in ymm3/m256 from packed unsigned word integers in ymm2 and saturate result.
EVEX.128.66.0F.WIG D8 /r VPSUBUSB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Subtract packed unsigned byte integers in xmm3/m128 from packed unsigned byte integers in xmm2, saturate results and store in xmm1 using writemask k1.
EVEX.256.66.0F.WIG D8 /r VPSUBUSB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Subtract packed unsigned byte integers in ymm3/m256 from packed unsigned byte integers in ymm2, saturate results and store in ymm1 using writemask k1.
EVEX.512.66.0F.WIG D8 /r VPSUBUSB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Subtract packed unsigned byte integers in zmm3/m512 from packed unsigned byte integers in zmm2, saturate results and store in zmm1 using writemask k1.
EVEX.128.66.0F.WIG D9 /r VPSUBUSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Subtract packed unsigned word integers in xmm3/m128 from packed unsigned word integers in xmm2 and saturate results and store in xmm1 using writemask k1.
EVEX.256.66.0F.WIG D9 /r VPSUBUSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Subtract packed unsigned word integers in ymm3/m256 from packed unsigned word integers in ymm2, saturate results and store in ymm1 using writemask k1.
EVEX.512.66.0F.WIG D9 /r VPSUBUSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Subtract packed unsigned word integers in zmm3/m512 from packed unsigned word integers in zmm2, saturate results and store in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD subtract of the packed unsigned integers of the source operand (second operand) from the packed unsigned integers of the destination operand (first operand), and stores the packed unsigned integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands.

The (V)PSUBUSB instruction subtracts packed unsigned byte integers. When an individual byte result is less than zero, the saturated value of 00H is written to the destination operand.

The (V)PSUBUSW instruction subtracts packed unsigned word integers. When an individual word result is less than zero, the saturated value of 0000H is written to the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PSUBUSB (With 64-bit Operands)

```
DEST[7:0] := SaturateToUnsignedByte (DEST[7:0] – SRC (7:0] );
(* Repeat add operation for 2nd through 7th bytes *)
DEST[63:56] := SaturateToUnsignedByte (DEST[63:56] – SRC[63:56];
```

PSUBUSW (With 64-bit Operands)

```
DEST[15:0] := SaturateToUnsignedWord (DEST[15:0] – SRC[15:0] );
(* Repeat add operation for 2nd and 3rd words *)
DEST[63:48] := SaturateToUnsignedWord (DEST[63:48] – SRC[63:48] );
```

VPSUBUSB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8;
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SaturateToUnsignedByte (SRC1[i+7:i] - SRC2[i+7:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] := 0;
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

VPSUBUSW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16;
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SaturateToUnsignedWord (SRC1[i+15:i] - SRC2[i+15:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] := 0;
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

VPSUBUSB (VEX.256 Encoded Version)

```
DEST[7:0] := SaturateToUnsignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 31st bytes *)
DEST[255:148] := SaturateToUnsignedByte (SRC1[255:248] - SRC2[255:248]);
DEST[MAXVL-1:256] := 0;
```

VPSUBUSB (VEX.128 Encoded Version)

```
DEST[7:0] := SaturateToUnsignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] := SaturateToUnsignedByte (SRC1[127:120] - SRC2[127:120]);
DEST[MAXVL-1:128] := 0
```

PSUBUSB (128-bit Legacy SSE Version)

```
DEST[7:0] := SaturateToUnsignedByte (DEST[7:0] - SRC[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] := SaturateToUnsignedByte (DEST[127:120] - SRC[127:120]);
DEST[MAXVL-1:128] (Unmodified)
```

VPSUBUSW (VEX.256 Encoded Version)
DEST[15:0] := SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 15th words *)
DEST[255:240] := SaturateToUnsignedWord (SRC1[255:240] - SRC2[255:240]);
DEST[MAXVL-1:256] := 0;

VPSUBUSW (VEX.128 Encoded Version)
DEST[15:0] := SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] := SaturateToUnsignedWord (SRC1[127:112] - SRC2[127:112]);
DEST[MAXVL-1:128] := 0

PSUBUSW (128-bit Legacy SSE Version)
DEST[15:0] := SaturateToUnsignedWord (DEST[15:0] - SRC[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] := SaturateToUnsignedWord (DEST[127:112] - SRC[127:112]);
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalents

VPSUBUSB __m512i _mm512_subs_epu8(__m512i a, __m512i b);
VPSUBUSB __m512i _mm512_mask_subs_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSUBUSB __m512i _mm512_maskz_subs_epu8(__mmask64 k, __m512i a, __m512i b);
VPSUBUSB __m256i _mm256_mask_subs_epu8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSUBUSB __m256i _mm256_maskz_subs_epu8(__mmask32 k, __m256i a, __m256i b);
VPSUBUSB __m128i _mm_mask_subs_epu8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSUBUSB __m128i _mm_maskz_subs_epu8(__mmask16 k, __m128i a, __m128i b);
VPSUBUSW __m512i _mm512_subs_epu16(__m512i a, __m512i b);
VPSUBUSW __m512i _mm512_mask_subs_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPSUBUSW __m512i _mm512_maskz_subs_epu16(__mmask32 k, __m512i a, __m512i b);
VPSUBUSW __m256i _mm256_mask_subs_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPSUBUSW __m256i _mm256_maskz_subs_epu16(__mmask16 k, __m256i a, __m256i b);
VPSUBUSW __m128i _mm_mask_subs_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBUSW __m128i _mm_maskz_subs_epu16(__mmask8 k, __m128i a, __m128i b);
PSUBUSB __m64 _mm_subs_pu8(__m64 m1, __m64 m2)
(V)PSUBUSB __m128i _mm_subs_epu8(__m128i m1, __m128i m2)
VPSUBUSB __m256i _mm256_subs_epu8(__m256i m1, __m256i m2)
PSUBUSW __m64 _mm_subs_pu16(__m64 m1, __m64 m2)
(V)PSUBUSW __m128i _mm_subs_epu16(__m128i m1, __m128i m2)
VPSUBUSW __m256i _mm256_subs_epu16(__m256i m1, __m256i m2)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-51, "Type E4 Class Exception Conditions."

PTEST—Logical Compare

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 17 /r PTEST xmm1, xmm2/m128	RM	V/V	SSE4_1	Set ZF if xmm2/m128 AND xmm1 result is all 0s. Set CF if xmm2/m128 AND NOT xmm1 result is all 0s.
VEX.128.66.0F38.WIG 17 /r VPTEST xmm1, xmm2/m128	RM	V/V	AVX	Set ZF and CF depending on bitwise AND and ANDN of sources.
VEX.256.66.0F38.WIG 17 /r VPTEST ymm1, ymm2/m256	RM	V/V	AVX	Set ZF and CF depending on bitwise AND and ANDN of sources.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	N/A	N/A

Description

PTEST and VPTEST set the ZF flag if all bits in the result are 0 of the bitwise AND of the first source operand (first operand) and the second source operand (second operand). VPTEST sets the CF flag if all bits in the result are 0 of the bitwise AND of the second source operand (second operand) and the logical NOT of the destination operand.

The first source register is specified by the ModR/M *reg* field.

128-bit versions: The first source register is an XMM register. The second source register can be an XMM register or a 128-bit memory location. The destination register is not modified.

VEX.256 encoded version: The first source register is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination register is not modified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

(V)PTEST (128-bit Version)

```
IF (SRC[127:0] BITWISE AND DEST[127:0] = 0)
    THEN ZF := 1;
    ELSE ZF := 0;
IF (SRC[127:0] BITWISE AND NOT DEST[127:0] = 0)
    THEN CF := 1;
    ELSE CF := 0;
DEST (unmodified)
AF := OF := PF := SF := 0;
```

VPTEST (VEX.256 Encoded Version)

```
IF (SRC[255:0] BITWISE AND DEST[255:0] = 0) THEN ZF := 1;
    ELSE ZF := 0;
IF (SRC[255:0] BITWISE AND NOT DEST[255:0] = 0) THEN CF := 1;
    ELSE CF := 0;
DEST (unmodified)
AF := OF := PF := SF := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

PTEST int _mm_testz_si128 (__m128i s1, __m128i s2);
PTEST int _mm_testc_si128 (__m128i s1, __m128i s2);
PTEST int _mm_testnzc_si128 (__m128i s1, __m128i s2);
VPTEST int _mm256_testz_si256 (__m256i s1, __m256i s2);
VPTEST int _mm256_testc_si256 (__m256i s1, __m256i s2);
VPTEST int _mm256_testnzc_si256 (__m256i s1, __m256i s2);
VPTEST int _mm_testz_si128 (__m128i s1, __m128i s2);
VPTEST int _mm_testc_si128 (__m128i s1, __m128i s2);
VPTEST int _mm_testnzc_si128 (__m128i s1, __m128i s2);

Flags Affected

The OF, AF, PF, SF flags are cleared and the ZF, CF flags are set according to the operation.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions," additionally:

#UD	If VEX.vvvv ≠ 1111B.

PTWRITE—Write Data to a Processor Trace Packet

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 REX.W 0F AE /4 PTWRITE r64/m64	RM	V/N.E	PTWRITE	Reads the data from r64/m64 to encode into a PTW packet if dependencies are met (see details below).
F3 0F AE /4 PTWRITE r32/m32	RM	V/V	PTWRITE	Reads the data from r32/m32 to encode into a PTW packet if dependencies are met (see details below).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:rm (r)	N/A	N/A	N/A

Description

This instruction reads data in the source operand and sends it to the Intel Processor Trace hardware to be encoded in a PTW packet if TriggerEn, ContextEn, FilterEn, and PTWEn are all set to 1. For more details on these values, see Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C, Section 36.2.2, "Software Trace Instrumentation with PTWRITE." The size of data is 64-bit if using REX.W in 64-bit mode, otherwise 32-bits of data are copied from the source operand.

Note: The instruction will #UD if prefix 66H is used.

Operation

```
IF (IA32_RTIT_STATUS.TriggerEn & IA32_RTIT_STATUS.ContextEn & IA32_RTIT_STATUS.FilterEn & IA32_RTIT_CTL.PTWEn) = 1
    PTW.PayloadBytes := Encoded payload size;
    PTW.IP := IA32_RTIT_CTL.FUPonPTW
    IF IA32_RTIT_CTL.FUPonPTW = 1
        Insert FUP packet with IP of PTWRITE;
    FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If CPUID.14H.00H:EBX.PTWRITE[4] = 0.
	If LOCK prefix is used.
	If 66H prefix is used.

Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CPUID.14H.00H:EBX.PTWRITE[4] = 0.
	If LOCK prefix is used.
	If 66H prefix is used.

Virtual 8086 Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If CPUID.14H.00H:EBX.PTWRITE[4] = 0.
	If LOCK prefix is used.
	If 66H prefix is used.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If CPUID.14H.00H:EBX.PTWRITE[4] = 0.
	If LOCK prefix is used.
	If 66H prefix is used.

PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ— Unpack High Data

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 68 /r[1] PUNPCKHBW mm, mm/m64	A	V/V	MMX	Unpack and interleave high-order bytes from mm and mm/m64 into mm.
66 0F 68 /r PUNPCKHBW xmm1, xmm2/m128	A	V/V	SSE2	Unpack and interleave high-order bytes from xmm1 and xmm2/m128 into xmm1.
NP 0F 69 /r[1] PUNPCKHWD mm, mm/m64	A	V/V	MMX	Unpack and interleave high-order words from mm and mm/m64 into mm.
66 0F 69 /r PUNPCKHWD xmm1, xmm2/m128	A	V/V	SSE2	Unpack and interleave high-order words from xmm1 and xmm2/m128 into xmm1.
NP 0F 6A /r[1] PUNPCKHDQ mm, mm/m64	A	V/V	MMX	Unpack and interleave high-order doublewords from mm and mm/m64 into mm.
66 0F 6A /r PUNPCKHDQ xmm1, xmm2/m128	A	V/V	SSE2	Unpack and interleave high-order doublewords from xmm1 and xmm2/m128 into xmm1.
66 0F 6D /r PUNPCKHQDQ xmm1, xmm2/m128	A	V/V	SSE2	Unpack and interleave high-order quadwords from xmm1 and xmm2/m128 into xmm1.
VEX.128.66.0F.WIG 68/r VPUNPCKHBW xmm1,xmm2, xmm3/m128	B	V/V	AVX	Interleave high-order bytes from xmm2 and xmm3/m128 into xmm1.
VEX.128.66.0F.WIG 69/r VPUNPCKHWD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Interleave high-order words from xmm2 and xmm3/m128 into xmm1.
VEX.128.66.0F.WIG 6A/r VPUNPCKHDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Interleave high-order doublewords from xmm2 and xmm3/m128 into xmm1.
VEX.128.66.0F.WIG 6D/r VPUNPCKHQDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Interleave high-order quadword from xmm2 and xmm3/m128 into xmm1 register.
VEX.256.66.0F.WIG 68 /r VPUNPCKHBW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave high-order bytes from ymm2 and ymm3/m256 into ymm1 register.
VEX.256.66.0F.WIG 69 /r VPUNPCKHWD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave high-order words from ymm2 and ymm3/m256 into ymm1 register.
VEX.256.66.0F.WIG 6A /r VPUNPCKHDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave high-order doublewords from ymm2 and ymm3/m256 into ymm1 register.
VEX.256.66.0F.WIG 6D /r VPUNPCKHQDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave high-order quadword from ymm2 and ymm3/m256 into ymm1 register.
EVEX.128.66.0F.WIG 68 /r VPUNPCKHBW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Interleave high-order bytes from xmm2 and xmm3/m128 into xmm1 register using k1 write mask.
EVEX.128.66.0F.WIG 69 /r VPUNPCKHWD xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Interleave high-order words from xmm2 and xmm3/m128 into xmm1 register using k1 write mask.
EVEX.128.66.0F.W0 6A /r VPUNPCKHDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Interleave high-order doublewords from xmm2 and xmm3/m128/m32bcst into xmm1 register using k1 write mask.
EVEX.128.66.0F.W1 6D /r VPUNPCKHQDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Interleave high-order quadword from xmm2 and xmm3/m128/m64bcst into xmm1 register using k1 write mask.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F.WIG 68 /r VPUNPCKHBW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Interleave high-order bytes from ymm2 and ymm3/m256 into ymm1 register using k1 write mask.
EVEX.256.66.0F.WIG 69 /r VPUNPCKHWD ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Interleave high-order words from ymm2 and ymm3/m256 into ymm1 register using k1 write mask.
EVEX.256.66.0F.W0 6A /r VPUNPCKHDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Interleave high-order doublewords from ymm2 and ymm3/m256/m32bcst into ymm1 register using k1 write mask.
EVEX.256.66.0F.W1 6D /r VPUNPCKHQDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Interleave high-order quadword from ymm2 and ymm3/m256/m64bcst into ymm1 register using k1 write mask.
EVEX.512.66.0F.WIG 68/r VPUNPCKHBW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Interleave high-order bytes from zmm2 and zmm3/m512 into zmm1 register.
EVEX.512.66.0F.WIG 69/r VPUNPCKHWD zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Interleave high-order words from zmm2 and zmm3/m512 into zmm1 register.
EVEX.512.66.0F.W0 6A /r VPUNPCKHDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F OR AVX10.1	Interleave high-order doublewords from zmm2 and zmm3/m512/m32bcst into zmm1 register using k1 write mask.
EVEX.512.66.0F.W1 6D /r VPUNPCKHQDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	D	V/V	AVX512F OR AVX10.1	Interleave high-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register using k1 write mask.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Unpacks and interleaves the high-order data elements (bytes, words, doublewords, or quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. Figure 4-20 shows the unpack operation for bytes in 64-bit operands. The low-order data elements are ignored.

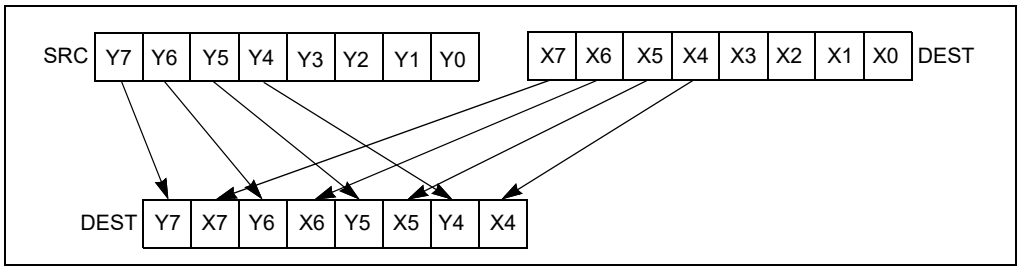


Figure 4-20. PUNPCKHBW Instruction Operation Using 64-bit Operands

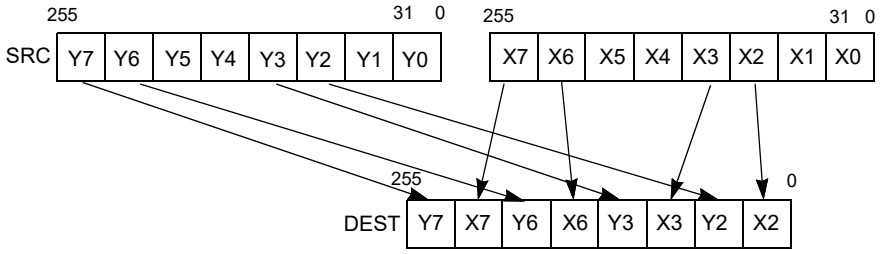


Figure 4-21. 256-bit VPUNPCKHDQ Instruction Operation

When the source data comes from a 64-bit memory operand, the full 64-bit operand is accessed from memory, but the instruction uses only the high-order 32 bits. When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The (V)PUNPCKHBW instruction interleaves the high-order bytes of the source and destination operands, the (V)PUNPCKHWD instruction interleaves the high-order words of the source and destination operands, the (V)PUNPCKHDQ instruction interleaves the high-order doubleword (or doublewords) of the source and destination operands, and the (V)PUNPCKHQDQ instruction interleaves the high-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the (V)PUNPCKHBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the (V)PUNPCKHWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE versions 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers.

EVEX encoded VPUNPCKHDQ/QDQ: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPUNPCKHWD/BW: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PUNPCKHBW Instruction With 64-bit Operands:

```
DEST[7:0] := DEST[39:32];
DEST[15:8] := SRC[39:32];
DEST[23:16] := DEST[47:40];
DEST[31:24] := SRC[47:40];
DEST[39:32] := DEST[55:48];
DEST[47:40] := SRC[55:48];
DEST[55:48] := DEST[63:56];
DEST[63:56] := SRC[63:56];
```

PUNPCKHW Instruction With 64-bit Operands:

```
DEST[15:0] := DEST[47:32];
DEST[31:16] := SRC[47:32];
DEST[47:32] := DEST[63:48];
DEST[63:48] := SRC[63:48];
```

PUNPCKHDQ Instruction With 64-bit Operands:

```
DEST[31:0] := DEST[63:32];
DEST[63:32] := SRC[63:32];
```

```
INTERLEAVE_HIGH_BYTES_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_HIGH_BYTES_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] := INTERLEAVE_HIGH_BYTES_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_HIGH_BYTES_256b (SRC1, SRC2)
DEST[7:0] := SRC1[71:64]
DEST[15:8] := SRC2[71:64]
DEST[23:16] := SRC1[79:72]
DEST[31:24] := SRC2[79:72]
DEST[39:32] := SRC1[87:80]
DEST[47:40] := SRC2[87:80]
DEST[55:48] := SRC1[95:88]
DEST[63:56] := SRC2[95:88]
DEST[71:64] := SRC1[103:96]
DEST[79:72] := SRC2[103:96]
DEST[87:80] := SRC1[111:104]
DEST[95:88] := SRC2[111:104]
DEST[103:96] := SRC1[119:112]
DEST[111:104] := SRC2[119:112]
DEST[119:112] := SRC1[127:120]
DEST[127:120] := SRC2[127:120]
DEST[135:128] := SRC1[199:192]
DEST[143:136] := SRC2[199:192]
DEST[151:144] := SRC1[207:200]
DEST[159:152] := SRC2[207:200]
```

```
DEST[167:160] := SRC1[215:208]
DEST[175:168] := SRC2[215:208]
DEST[183:176] := SRC1[223:216]
DEST[191:184] := SRC2[223:216]
DEST[199:192] := SRC1[231:224]
DEST[207:200] := SRC2[231:224]
DEST[215:208] := SRC1[239:232]
DEST[223:216] := SRC2[239:232]
DEST[231:224] := SRC1[247:240]
DEST[239:232] := SRC2[247:240]
DEST[247:240] := SRC1[255:248]
DEST[255:248] := SRC2[255:248]

INTERLEAVE_HIGH_BYTES (SRC1, SRC2)
DEST[7:0] := SRC1[71:64]
DEST[15:8] := SRC2[71:64]
DEST[23:16] := SRC1[79:72]
DEST[31:24] := SRC2[79:72]
DEST[39:32] := SRC1[87:80]
DEST[47:40] := SRC2[87:80]
DEST[55:48] := SRC1[95:88]
DEST[63:56] := SRC2[95:88]
DEST[71:64] := SRC1[103:96]
DEST[79:72] := SRC2[103:96]
DEST[87:80] := SRC1[111:104]
DEST[95:88] := SRC2[111:104]
DEST[103:96] := SRC1[119:112]
DEST[111:104] := SRC2[119:112]
DEST[119:112] := SRC1[127:120]
DEST[127:120] := SRC2[127:120]

INTERLEAVE_HIGH_WORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_HIGH_WORDS_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] := INTERLEAVE_HIGH_WORDS_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_HIGH_WORDS_256b(SRC1, SRC2)
DEST[15:0] := SRC1[79:64]
DEST[31:16] := SRC2[79:64]
DEST[47:32] := SRC1[95:80]
DEST[63:48] := SRC2[95:80]
DEST[79:64] := SRC1[111:96]
DEST[95:80] := SRC2[111:96]
DEST[111:96] := SRC1[127:112]
DEST[127:112] := SRC2[127:112]
DEST[143:128] := SRC1[207:192]
DEST[159:144] := SRC2[207:192]
DEST[175:160] := SRC1[223:208]
DEST[191:176] := SRC2[223:208]
DEST[207:192] := SRC1[239:224]
DEST[223:208] := SRC2[239:224]
DEST[239:224] := SRC1[255:240]
DEST[255:240] := SRC2[255:240]

INTERLEAVE_HIGH_WORDS (SRC1, SRC2)
```

```
DEST[15:0] := SRC1[79:64]
DEST[31:16] := SRC2[79:64]
DEST[47:32] := SRC1[95:80]
DEST[63:48] := SRC2[95:80]
DEST[79:64] := SRC1[111:96]
DEST[95:80] := SRC2[111:96]
DEST[111:96] := SRC1[127:112]
DEST[127:112] := SRC2[127:112]

INTERLEAVE_HIGH_DWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_HIGH_DWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] := INTERLEAVE_HIGH_DWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_HIGH_DWORDS_256b(SRC1, SRC2)
DEST[31:0] := SRC1[95:64]
DEST[63:32] := SRC2[95:64]
DEST[95:64] := SRC1[127:96]
DEST[127:96] := SRC2[127:96]
DEST[159:128] := SRC1[223:192]
DEST[191:160] := SRC2[223:192]
DEST[223:192] := SRC1[255:224]
DEST[255:224] := SRC2[255:224]

INTERLEAVE_HIGH_DWORDS(SRC1, SRC2)
DEST[31:0] := SRC1[95:64]
DEST[63:32] := SRC2[95:64]
DEST[95:64] := SRC1[127:96]
DEST[127:96] := SRC2[127:96]

INTERLEAVE_HIGH_QWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_HIGH_QWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] := INTERLEAVE_HIGH_QWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_HIGH_QWORDS_256b(SRC1, SRC2)
DEST[63:0] := SRC1[127:64]
DEST[127:64] := SRC2[127:64]
DEST[191:128] := SRC1[255:192]
DEST[255:192] := SRC2[255:192]

INTERLEAVE_HIGH_QWORDS(SRC1, SRC2)
DEST[63:0] := SRC1[127:64]
DEST[127:64] := SRC2[127:64]
```

PUNPCKHBW (128-bit Legacy SSE Version)

```
DEST[127:0] := INTERLEAVE_HIGH_BYTES(DEST, SRC)
DEST[255:127] (Unmodified)
```

VPUNPCKHBW (VEX.128 Encoded Version)

```
DEST[127:0] := INTERLEAVE_HIGH_BYTES(SRC1, SRC2)
DEST[MAXVL-1:127] := 0
```

VPUNPCKHBW (VEX.256 Encoded Version)

```
DEST[255:0] := INTERLEAVE_HIGH_BYTES_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPUNPCKHBW (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
IF VL = 128
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_BYTES(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_BYTES_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_BYTES_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TMP_DEST[i+7:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

PUNPCKHWD (128-bit Legacy SSE Version)

```
DEST[127:0] := INTERLEAVE_HIGH_WORDS(DEST, SRC)
DEST[255:127] (Unmodified)
```

VPUNPCKHWD (VEX.128 Encoded Version)

```
DEST[127:0] := INTERLEAVE_HIGH_WORDS(SRC1, SRC2)
DEST[MAXVL-1:127] := 0
```

VPUNPCKHWD (VEX.256 Encoded Version)

```
DEST[255:0] := INTERLEAVE_HIGH_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPUNPCKHWD (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_WORDS(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_WORDS_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_WORDS_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
```

```
        THEN DEST[i+15:i] := TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

PUNPCKHDQ (128-bit Legacy SSE Version)

```
DEST[127:0] := INTERLEAVE_HIGH_DWORDS(DEST, SRC)
DEST[255:127] (Unmodified)
```

VPUNPCKHDQ (VEX.128 Encoded Version)

```
DEST[127:0] := INTERLEAVE_HIGH_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:127] := 0
```

VPUNPCKHDQ (VEX.256 Encoded Version)

```
DEST[255:0] := INTERLEAVE_HIGH_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPUNPCKHDQ (EVEX.512 Encoded Version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] := SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_DWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_DWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_DWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
```

```
DEST[MAXVL-1:VL] := 0
```

PUNPCKHQDQ (128-bit Legacy SSE Version)

```
DEST[127:0] := INTERLEAVE_HIGH_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

VPUNPCKHQDQ (VEX.128 Encoded Version)

```
DEST[127:0] := INTERLEAVE_HIGH_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPUNPCKHQDQ (VEX.256 Encoded Version)

```
DEST[255:0] := INTERLEAVE_HIGH_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPUNPCKHQDQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_QWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_QWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_QWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPUNPCKHBW __m512i _mm512_unpackhi_epi8(__m512i a, __m512i b);
VPUNPCKHBW __m512i _mm512_mask_unpackhi_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPUNPCKHBW __m512i _mm512_maskz_unpackhi_epi8( __mmask64 k, __m512i a, __m512i b);
VPUNPCKHBW __m256i _mm256_mask_unpackhi_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPUNPCKHBW __m256i _mm256_maskz_unpackhi_epi8( __mmask32 k, __m256i a, __m256i b);
```

VPUNPCKHBW __m128i _mm_mask_unpackhi_epi8(v s, __mmask16 k, __m128i a, __m128i b);
VPUNPCKHBW __m128i _mm_maskz_unpackhi_epi8(__mmask16 k, __m128i a, __m128i b);
VPUNPCKHWD __m512i _mm512_unpackhi_epi16(__m512i a, __m512i b);
VPUNPCKHWD __m512i _mm512_mask_unpackhi_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPUNPCKHWD __m512i _mm512_maskz_unpackhi_epi16(__mmask32 k, __m512i a, __m512i b);
VPUNPCKHWD __m256i _mm256_mask_unpackhi_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPUNPCKHWD __m256i _mm256_maskz_unpackhi_epi16(__mmask16 k, __m256i a, __m256i b);
VPUNPCKHWD __m128i _mm_mask_unpackhi_epi16(v s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKHWD __m128i _mm_maskz_unpackhi_epi16(__mmask8 k, __m128i a, __m128i b);
VPUNPCKHDQ __m512i _mm512_unpackhi_epi32(__m512i a, __m512i b);
VPUNPCKHDQ __m512i _mm512_mask_unpackhi_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPUNPCKHDQ __m512i _mm512_maskz_unpackhi_epi32(__mmask16 k, __m512i a, __m512i b);
VPUNPCKHDQ __m256i _mm256_mask_unpackhi_epi32(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHDQ __m256i _mm256_maskz_unpackhi_epi32(__mmask8 k, __m512i a, __m512i b);
VPUNPCKHDQ __m128i _mm_mask_unpackhi_epi32(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHDQ __m128i _mm_maskz_unpackhi_epi32(__mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m512i _mm512_unpackhi_epi64(__m512i a, __m512i b);
VPUNPCKHQDQ __m512i _mm512_mask_unpackhi_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m512i _mm512_maskz_unpackhi_epi64(__mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m256i _mm256_mask_unpackhi_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m256i _mm256_maskz_unpackhi_epi64(__mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m128i _mm_mask_unpackhi_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m128i _mm_maskz_unpackhi_epi64(__mmask8 k, __m512i a, __m512i b);
PUNPCKHBW __m64 _mm_unpackhi_pi8(__m64 m1, __m64 m2)
(V)PUNPCKHBW __m128i _mm_unpackhi_epi8(__m128i m1, __m128i m2)
VPUNPCKHBW __m256i _mm256_unpackhi_epi8(__m256i m1, __m256i m2)
PUNPCKHWD __m64 _mm_unpackhi_pi16(__m64 m1,__m64 m2)
(V)PUNPCKHWD __m128i _mm_unpackhi_epi16(__m128i m1,__m128i m2)
VPUNPCKHWD __m256i _mm256_unpackhi_epi16(__m256i m1,__m256i m2)
PUNPCKHDQ __m64 _mm_unpackhi_pi32(__m64 m1, __m64 m2)
(V)PUNPCKHDQ __m128i _mm_unpackhi_epi32(__m128i m1, __m128i m2)
VPUNPCKHDQ __m256i _mm256_unpackhi_epi32(__m256i m1, __m256i m2)
(V)PUNPCKHQDQ __m128i _mm_unpackhi_epi64 (__m128i a, __m128i b)
VPUNPCKHQDQ __m256i _mm256_unpackhi_epi64 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."

EVEX-encoded VPUNPCKHQDQ/QDQ, see Table 2-52, "Type E4NF Class Exception Conditions."

EVEX-encoded VPUNPCKHBW/WD, see Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."

PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ—Unpack Low Data

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 60 /r[1] PUNPCKLBW mm, mm/m32	A	V/V	MMX	Interleave low-order bytes from mm and mm/m32 into mm.
66 0F 60 /r PUNPCKLBW xmm1, xmm2/m128	A	V/V	SSE2	Interleave low-order bytes from xmm1 and xmm2/m128 into xmm1.
NP 0F 61 /r[1] PUNPCKLWD mm, mm/m32	A	V/V	MMX	Interleave low-order words from mm and mm/m32 into mm.
66 0F 61 /r PUNPCKLWD xmm1, xmm2/m128	A	V/V	SSE2	Interleave low-order words from xmm1 and xmm2/m128 into xmm1.
NP 0F 62 /r[1] PUNPCKLDQ mm, mm/m32	A	V/V	MMX	Interleave low-order doublewords from mm and mm/m32 into mm.
66 0F 62 /r PUNPCKLDQ xmm1, xmm2/m128	A	V/V	SSE2	Interleave low-order doublewords from xmm1 and xmm2/m128 into xmm1.
66 0F 6C /r PUNPCKLQDQ xmm1, xmm2/m128	A	V/V	SSE2	Interleave low-order quadword from xmm1 and xmm2/m128 into xmm1 register.
VEX.128.66.0F.WIG 60/r VPUNPCKLBW xmm1,xmm2, xmm3/m128	B	V/V	AVX	Interleave low-order bytes from xmm2 and xmm3/m128 into xmm1.
VEX.128.66.0F.WIG 61/r VPUNPCKLWD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Interleave low-order words from xmm2 and xmm3/m128 into xmm1.
VEX.128.66.0F.WIG 62/r VPUNPCKLDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Interleave low-order doublewords from xmm2 and xmm3/m128 into xmm1.
VEX.128.66.0F.WIG 6C/r VPUNPCKLQDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Interleave low-order quadword from xmm2 and xmm3/m128 into xmm1 register.
VEX.256.66.0F.WIG 60 /r VPUNPCKLBW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave low-order bytes from ymm2 and ymm3/m256 into ymm1 register.
VEX.256.66.0F.WIG 61 /r VPUNPCKLWD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave low-order words from ymm2 and ymm3/m256 into ymm1 register.
VEX.256.66.0F.WIG 62 /r VPUNPCKLDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave low-order doublewords from ymm2 and ymm3/m256 into ymm1 register.
VEX.256.66.0F.WIG 6C /r VPUNPCKLQDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave low-order quadword from ymm2 and ymm3/m256 into ymm1 register.
EVEX.128.66.0F.WIG 60 /r VPUNPCKLBW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Interleave low-order bytes from xmm2 and xmm3/m128 into xmm1 register subject to write mask k1.
EVEX.128.66.0F.WIG 61 /r VPUNPCKLWD xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Interleave low-order words from xmm2 and xmm3/m128 into xmm1 register subject to write mask k1.
EVEX.128.66.0F.W0 62 /r VPUNPCKLDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Interleave low-order doublewords from xmm2 and xmm3/m128/m32bcst into xmm1 register subject to write mask k1.
EVEX.128.66.0F.W1 6C /r VPUNPCKLQDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Interleave low-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register subject to write mask k1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F.WIG 60 /r VPUNPCKLBW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Interleave low-order bytes from ymm2 and ymm3/m256 into ymm1 register subject to write mask k1.
EVEX.256.66.0F.WIG 61 /r VPUNPCKLWD ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Interleave low-order words from ymm2 and ymm3/m256 into ymm1 register subject to write mask k1.
EVEX.256.66.0F.W0 62 /r VPUNPCKLDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Interleave low-order doublewords from ymm2 and ymm3/m256/m32bcst into ymm1 register subject to write mask k1.
EVEX.256.66.0F.W1 6C /r VPUNPCKLQDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Interleave low-order quadword from ymm2 and ymm3/m256/m64bcst into ymm1 register subject to write mask k1.
EVEX.512.66.0F.WIG 60/r VPUNPCKLBW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Interleave low-order bytes from zmm2 and zmm3/m512 into zmm1 register subject to write mask k1.
EVEX.512.66.0F.WIG 61/r VPUNPCKLWD zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Interleave low-order words from zmm2 and zmm3/m512 into zmm1 register subject to write mask k1.
EVEX.512.66.0F.W0 62 /r VPUNPCKLDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F OR AVX10.1	Interleave low-order doublewords from zmm2 and zmm3/m512/m32bcst into zmm1 register subject to write mask k1.
EVEX.512.66.0F.W1 6C /r VPUNPCKLQDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	D	V/V	AVX512F OR AVX10.1	Interleave low-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register subject to write mask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Unpacks and interleaves the low-order data elements (bytes, words, doublewords, and quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. (Figure 4-22 shows the unpack operation for bytes in 64-bit operands.). The high-order data elements are ignored.

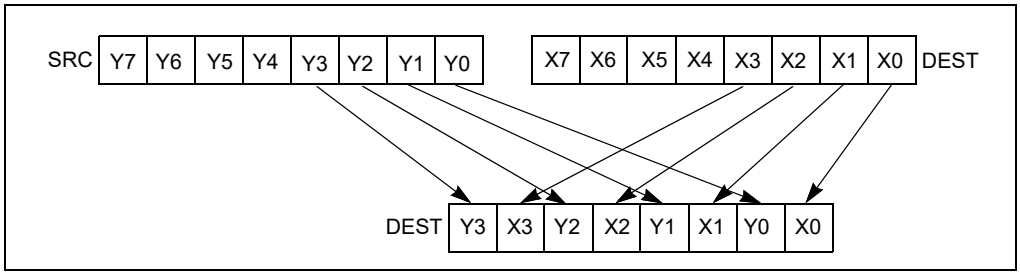


Figure 4-22. PUNPCKLBW Instruction Operation Using 64-bit Operands

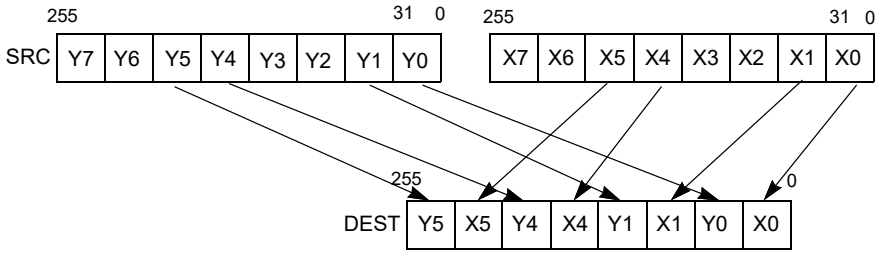


Figure 4-23. 256-bit VPUNPCKLDQ Instruction Operation

When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The (V)PUNPCKLBW instruction interleaves the low-order bytes of the source and destination operands, the (V)PUNPCKLWD instruction interleaves the low-order words of the source and destination operands, the (V)PUNPCKLDQ instruction interleaves the low-order doubleword (or doublewords) of the source and destination operands, and the (V)PUNPCKLQDQ instruction interleaves the low-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the (V)PUNPCKLBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the (V)PUNPCKLWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE versions 64-bit operand: The source operand can be an MMX technology register or a 32-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPUNPCKLDQ/QDQ: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source

operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPUNPCKLWD/BW: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PUNPCKLBW Instruction With 64-bit Operands:

```
    DEST[63:56] := SRC[31:24];
    DEST[55:48] := DEST[31:24];
    DEST[47:40] := SRC[23:16];
    DEST[39:32] := DEST[23:16];
    DEST[31:24] := SRC[15:8];
    DEST[23:16] := DEST[15:8];
    DEST[15:8] := SRC[7:0];
    DEST[7:0] := DEST[7:0];
```

PUNPCKLWD Instruction With 64-bit Operands:

```
    DEST[63:48] := SRC[31:16];
    DEST[47:32] := DEST[31:16];
    DEST[31:16] := SRC[15:0];
    DEST[15:0] := DEST[15:0];
```

PUNPCKLDQ Instruction With 64-bit Operands:

```
    DEST[63:32] := SRC[31:0];
    DEST[31:0] := DEST[31:0];
INTERLEAVE_BYTES_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_BYTES_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] := INTERLEAVE_BYTES_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_BYTES_256b (SRC1, SRC2)
DEST[7:0] := SRC1[7:0]
DEST[15:8] := SRC2[7:0]
DEST[23:16] := SRC1[15:8]
DEST[31:24] := SRC2[15:8]
DEST[39:32] := SRC1[23:16]
DEST[47:40] := SRC2[23:16]
DEST[55:48] := SRC1[31:24]
DEST[63:56] := SRC2[31:24]
DEST[71:64] := SRC1[39:32]
DEST[79:72] := SRC2[39:32]
DEST[87:80] := SRC1[47:40]
DEST[95:88] := SRC2[47:40]
DEST[103:96] := SRC1[55:48]
DEST[111:104] := SRC2[55:48]
DEST[119:112] := SRC1[63:56]
DEST[127:120] := SRC2[63:56]
DEST[135:128] := SRC1[135:128]
DEST[143:136] := SRC2[135:128]
DEST[151:144] := SRC1[143:136]
DEST[159:152] := SRC2[143:136]
DEST[167:160] := SRC1[151:144]
DEST[175:168] := SRC2[151:144]
```

```
DEST[183:176] := SRC1[159:152]
DEST[191:184] := SRC2[159:152]
DEST[199:192] := SRC1[167:160]
DEST[207:200] := SRC2[167:160]
DEST[215:208] := SRC1[175:168]
DEST[223:216] := SRC2[175:168]
DEST[231:224] := SRC1[183:176]
DEST[239:232] := SRC2[183:176]
DEST[247:240] := SRC1[191:184]
DEST[255:248] := SRC2[191:184]

INTERLEAVE_BYTES (SRC1, SRC2)
DEST[7:0] := SRC1[7:0]
DEST[15:8] := SRC2[7:0]
DEST[23:16] := 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]

INTERLEAVE_WORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_WORDS_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] := INTERLEAVE_WORDS_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_WORDS_256b(SRC1, SRC2)
DEST[15:0] := SRC1[15:0]
DEST[31:16] := SRC2[15:0]
DEST[47:32] := SRC1[31:16]
DEST[63:48] := SRC2[31:16]
DEST[79:64] := SRC1[47:32]
DEST[95:80] := SRC2[47:32]
DEST[111:96] := SRC1[63:48]
DEST[127:112] := SRC2[63:48]
DEST[143:128] := SRC1[143:128]
DEST[159:144] := SRC2[143:128]
DEST[175:160] := SRC1[159:144]
DEST[191:176] := SRC2[159:144]
DEST[207:192] := SRC1[175:160]
DEST[223:208] := SRC2[175:160]
DEST[239:224] := SRC1[191:176]
DEST[255:240] := SRC2[191:176]

INTERLEAVE_WORDS (SRC1, SRC2)
DEST[15:0] := SRC1[15:0]
DEST[31:16] := SRC2[15:0]
```

```
DEST[47:32] := SRC1[31:16]
DEST[63:48] := SRC2[31:16]
DEST[79:64] := SRC1[47:32]
DEST[95:80] := SRC2[47:32]
DEST[111:96] := SRC1[63:48]
DEST[127:112] := SRC2[63:48]

INTERLEAVE_DWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_DWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] := INTERLEAVE_DWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_DWORDS_256b(SRC1, SRC2)
DEST[31:0] := SRC1[31:0]
DEST[63:32] := SRC2[31:0]
DEST[95:64] := SRC1[63:32]
DEST[127:96] := SRC2[63:32]
DEST[159:128] := SRC1[159:128]
DEST[191:160] := SRC2[159:128]
DEST[223:192] := SRC1[191:160]
DEST[255:224] := SRC2[191:160]

INTERLEAVE_DWORDS(SRC1, SRC2)
DEST[31:0] := SRC1[31:0]
DEST[63:32] := SRC2[31:0]
DEST[95:64] := SRC1[63:32]
DEST[127:96] := SRC2[63:32]
INTERLEAVE_QWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_QWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] := INTERLEAVE_QWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_QWORDS_256b(SRC1, SRC2)
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[191:128] := SRC1[191:128]
DEST[255:192] := SRC2[191:128]

INTERLEAVE_QWORDS(SRC1, SRC2)
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
```

PUNPCKLBW

```
DEST[127:0] := INTERLEAVE_BYTES(DEST, SRC)
DEST[255:127] (Unmodified)
```

VPUNPCKLBW (VEX.128 Encoded Instruction)

```
DEST[127:0] := INTERLEAVE_BYTES(SRC1, SRC2)
DEST[MAXVL-1:127] := 0
```

VPUNPCKLBW (VEX.256 Encoded Instruction)

```
DEST[255:0] := INTERLEAVE_BYTES_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPUNPCKLBW (EVEX.512 Encoded Instruction)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
IF VL = 128
    TMP_DEST[VL-1:0] := INTERLEAVE_BYTES(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] := INTERLEAVE_BYTES_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] := INTERLEAVE_BYTES_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TMP_DEST[i+7:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
DEST[511:0] := INTERLEAVE_BYTES_512b(SRC1, SRC2)
```

PUNPCKLWD

```
DEST[127:0] := INTERLEAVE_WORDS(DEST, SRC)
DEST[255:127] (Unmodified)
```

VPUNPCKLWD (VEX.128 Encoded Instruction)

```
DEST[127:0] := INTERLEAVE_WORDS(SRC1, SRC2)
DEST[MAXVL-1:127] := 0
```

VPUNPCKLWD (VEX.256 Encoded Instruction)

```
DEST[255:0] := INTERLEAVE_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPUNPCKLWD (EVEX.512 Encoded Instruction)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[VL-1:0] := INTERLEAVE_WORDS(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] := INTERLEAVE_WORDS_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] := INTERLEAVE_WORDS_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
```

```
        THEN DEST[i+15:i] := TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
DEST[511:0] := INTERLEAVE_WORDS_512b(SRC1, SRC2)
```

PUNPCKLDQ

```
DEST[127:0] := INTERLEAVE_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

VPUNPCKLDQ (VEX.128 Encoded Instruction)

```
DEST[127:0] := INTERLEAVE_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPUNPCKLDQ (VEX.256 Encoded Instruction)

```
DEST[255:0] := INTERLEAVE_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPUNPCKLDQ (EVEX Encoded Instructions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] := SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] := INTERLEAVE_DWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] := INTERLEAVE_DWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] := INTERLEAVE_DWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
```

```
ENDFOR
DEST511:0] := INTERLEAVE_DWORDS_512b(SRC1, SRC2)
DEST[MAXVL-1:VL] := 0
```

PUNPCKLQDQ

```
DEST[127:0] := INTERLEAVE_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

VPUNPCKLQDQ (VEX.128 Encoded Instruction)

```
DEST[127:0] := INTERLEAVE_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPUNPCKLQDQ (VEX.256 Encoded Instruction)

```
DEST[255:0] := INTERLEAVE_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPUNPCKLQDQ (EVEX Encoded Instructions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] := INTERLEAVE_QWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] := INTERLEAVE_QWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] := INTERLEAVE_QWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPUNPCKLBW __m512i _mm512_unpacklo_epi8(__m512i a, __m512i b);
VPUNPCKLBW __m512i _mm512_mask_unpacklo_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPUNPCKLBW __m512i _mm512_maskz_unpacklo_epi8( __mmask64 k, __m512i a, __m512i b);
```

VPUNPCKLBW __m256i _mm256_mask_unpacklo_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPUNPCKLBW __m256i _mm256_maskz_unpacklo_epi8(__mmask32 k, __m256i a, __m256i b);
VPUNPCKLBW __m128i _mm_mask_unpacklo_epi8(v s, __mmask16 k, __m128i a, __m128i b);
VPUNPCKLBW __m128i _mm_maskz_unpacklo_epi8(__mmask16 k, __m128i a, __m128i b);
VPUNPCKLWD __m512i _mm512_unpacklo_epi16(__m512i a, __m512i b);
VPUNPCKLWD __m512i _mm512_mask_unpacklo_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPUNPCKLWD __m512i _mm512_maskz_unpacklo_epi16(__mmask32 k, __m512i a, __m512i b);
VPUNPCKLWD __m256i _mm256_mask_unpacklo_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPUNPCKLWD __m256i _mm256_maskz_unpacklo_epi16(__mmask16 k, __m256i a, __m256i b);
VPUNPCKLWD __m128i _mm_mask_unpacklo_epi16(v s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKLWD __m128i _mm_maskz_unpacklo_epi16(__mmask8 k, __m128i a, __m128i b);
VPUNPCKLDQ __m512i _mm512_unpacklo_epi32(__m512i a, __m512i b);
VPUNPCKLDQ __m512i _mm512_mask_unpacklo_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPUNPCKLDQ __m512i _mm512_maskz_unpacklo_epi32(__mmask16 k, __m512i a, __m512i b);
VPUNPCKLDQ __m256i _mm256_mask_unpacklo_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPUNPCKLDQ __m256i _mm256_maskz_unpacklo_epi32(__mmask8 k, __m256i a, __m256i b);
VPUNPCKLDQ __m128i _mm_mask_unpacklo_epi32(v s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKLDQ __m128i _mm_maskz_unpacklo_epi32(__mmask8 k, __m128i a, __m128i b);
VPUNPCKLQDQ __m512i _mm512_unpacklo_epi64(__m512i a, __m512i b);
VPUNPCKLQDQ __m512i _mm512_mask_unpacklo_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKLQDQ __m512i _mm512_maskz_unpacklo_epi64(__mmask8 k, __m512i a, __m512i b);
VPUNPCKLQDQ __m256i _mm256_mask_unpacklo_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPUNPCKLQDQ __m256i _mm256_maskz_unpacklo_epi64(__mmask8 k, __m256i a, __m256i b);
VPUNPCKLQDQ __m128i _mm_mask_unpacklo_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKLQDQ __m128i _mm_maskz_unpacklo_epi64(__mmask8 k, __m128i a, __m128i b);
PUNPCKLBW __m64 _mm_unpacklo_pi8 (__m64 m1, __m64 m2)
(V)PUNPCKLBW __m128i _mm_unpacklo_epi8 (__m128i m1, __m128i m2)
VPUNPCKLBW __m256i _mm256_unpacklo_epi8 (__m256i m1, __m256i m2)
PUNPCKLWD __m64 _mm_unpacklo_pi16 (__m64 m1, __m64 m2)
(V)PUNPCKLWD __m128i _mm_unpacklo_epi16 (__m128i m1, __m128i m2)
VPUNPCKLWD __m256i _mm256_unpacklo_epi16 (__m256i m1, __m256i m2)
PUNPCKLDQ __m64 _mm_unpacklo_pi32 (__m64 m1, __m64 m2)
(V)PUNPCKLDQ __m128i _mm_unpacklo_epi32 (__m128i m1, __m128i m2)
VPUNPCKLDQ __m256i _mm256_unpacklo_epi32 (__m256i m1, __m256i m2)
(V)PUNPCKLQDQ __m128i _mm_unpacklo_epi64 (__m128i m1, __m128i m2)
VPUNPCKLQDQ __m256i _mm256_unpacklo_epi64 (__m256i m1, __m256i m2)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."

EVEX-encoded VPUNPCKLDQ/QDQ, see Table 2-52, "Type E4NF Class Exception Conditions."

EVEX-encoded VPUNPCKLBW/WD, see Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."

PUSH—Push Word, Doubleword, or Quadword Onto the Stack

Opcode[1]	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FF /6	PUSH r/m16	M	Valid	Valid	Push r/m16.
FF /6	PUSH r/m32	M	N.E.	Valid	Push r/m32.
FF /6	PUSH r/m64	M	Valid	N.E.	Push r/m64.
50+rw	PUSH r16	O	Valid	Valid	Push r16.
50+rd	PUSH r32	O	N.E.	Valid	Push r32.
50+rd	PUSH r64	O	Valid	N.E.	Push r64.
6A ib	PUSH imm8	I	Valid	Valid	Push imm8.
68 iw	PUSH imm16	I	Valid	Valid	Push imm16.
68 id	PUSH imm32	I	Valid	Valid	Push imm32.
0E	PUSH CS	ZO	Invalid	Valid	Push CS.
16	PUSH SS	ZO	Invalid	Valid	Push SS.
1E	PUSH DS	ZO	Invalid	Valid	Push DS.
06	PUSH ES	ZO	Invalid	Valid	Push ES.
0F A0	PUSH FS	ZO	Valid	Valid	Push FS.
0F A8	PUSH GS	ZO	Valid	Valid	Push GS.

NOTES:

1. See the IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A
O	opcode + rd (r)	N/A	N/A	N/A
I	imm8/16/32	N/A	N/A	N/A
ZO	N/A	N/A	N/A	N/A

Description

Decrements the stack pointer and then stores the source operand on the top of the stack. Address and operand sizes are determined and used as follows:

- Address size. The D flag in the current code-segment descriptor determines the default address size; it may be overridden by an instruction prefix (67H).

  The address size is used only when referencing a source operand in memory.

- Operand size. The D flag in the current code-segment descriptor determines the default operand size; it may be overridden by instruction prefixes (66H or REX.W).

  The operand size (16, 32, or 64 bits) determines the amount by which the stack pointer is decremented (2, 4 or 8).

  If the source operand is an immediate of size less than the operand size, a sign-extended value is pushed on the stack. If the source operand is a segment register (16 bits) and the operand size is 64-bits, a zero-extended value is pushed on the stack; if the operand size is 32-bits, either a zero-extended value is pushed on the stack or the segment selector is written on the stack using a 16-bit move. For the last case, all recent Intel Core and Intel Atom processors perform a 16-bit move, leaving the upper portion of the stack location unmodified.

- Stack-address size. Outside of 64-bit mode, the B flag in the current stack-segment descriptor determines the size of the stack pointer (16 or 32 bits); in 64-bit mode, the size of the stack pointer is always 64 bits.

  The stack-address size determines the width of the stack pointer when writing to the stack in memory and when decrementing the stack pointer. (As stated above, the amount by which the stack pointer is decremented is determined by the operand size.)

  If the operand size is less than the stack-address size, the PUSH instruction may result in a misaligned stack pointer (a stack pointer that is not aligned on a doubleword or quadword boundary).

The PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. If a PUSH instruction uses a memory operand in which the ESP register is used for computing the operand address, the address of the operand is computed before the ESP register is decremented.

If the ESP or SP register is 1 when the PUSH instruction is executed in real-address mode, a stack-fault exception (#SS) is generated (because the limit of the stack segment is violated). Its delivery encounters a second stack-fault exception (for the same reason), causing generation of a double-fault exception (#DF). Delivery of the double-fault exception encounters a third stack-fault exception, and the logical processor enters shutdown mode. See the discussion of the double-fault exception in Chapter 7 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

IA-32 Architecture Compatibility

For IA-32 processors from the Intel 286 on, the PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. (This is also true for Intel 64 architecture, real-address and virtual-8086 modes of IA-32 architecture.) For the Intel® 8086 processor, the PUSH SP instruction pushes the new value of the SP register (that is the value after it has been decremented by 2).

Operation

```
(* See Description section for possible sign-extension or zero-extension of source operand and for *)
(* a case in which the size of the memory store may be smaller than the instruction's operand size *)
IF StackAddrSize = 64
    THEN
        IF OperandSize = 64
            THEN
                RSP := RSP – 8;
                Memory[SS:RSP] := SRC;          (* push quadword *)
        ELSE IF OperandSize = 32
            THEN
                RSP := RSP – 4;
                Memory[SS:RSP] := SRC;          (* push dword *)
            ELSE (* OperandSize = 16 *)
                RSP := RSP – 2;
                Memory[SS:RSP] := SRC;          (* push word *)
        FI;
ELSE IF StackAddrSize = 32
    THEN
        IF OperandSize = 64
            THEN
                ESP := ESP – 8;
                Memory[SS:ESP] := SRC;          (* push quadword *)
        ELSE IF OperandSize = 32
            THEN
                ESP := ESP – 4;
                Memory[SS:ESP] := SRC;          (* push dword *)
            ELSE (* OperandSize = 16 *)
                ESP := ESP – 2;
                Memory[SS:ESP] := SRC;          (* push word *)
```

```
        FI;
    ELSE (* StackAddrSize = 16 *)
        IF OperandSize = 32
            THEN
                SP := SP – 4;
                Memory[SS:SP] := SRC;           (* push dword *)
            ELSE (* OperandSize = 16 *)
                SP := SP – 2;
                Memory[SS:SP] := SRC;           (* push word *)
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
	If the new value of the SP or ESP register is outside the stack segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.
	If the PUSH is of CS, SS, DS, or ES.

PUSHA/PUSHAD—Push All General-Purpose Registers

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
60	PUSHA	ZO	Invalid	Valid	Push AX, CX, DX, BX, original SP, BP, SI, and DI.
60	PUSHAD	ZO	Invalid	Valid	Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Pushes the contents of the general-purpose registers onto the stack. The registers are stored on the stack in the following order: EAX, ECX, EDX, EBX, ESP (original value), EBP, ESI, and EDI (if the current operand-size attribute is 32) and AX, CX, DX, BX, SP (original value), BP, SI, and DI (if the operand-size attribute is 16). These instructions perform the reverse operation of the POPA/POPAD instructions. The value pushed for the ESP or SP register is its value before prior to pushing the first register (see the "Operation" section below).

The PUSHA (push all) and PUSHAD (push all double) mnemonics reference the same opcode. The PUSHA instruction is intended for use when the operand-size attribute is 16 and the PUSHAD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when PUSHA is used and to 32 when PUSHAD is used. Others may treat these mnemonics as synonyms (PUSHA/PUSHAD) and use the current setting of the operand-size attribute to determine the size of values to be pushed from the stack, regardless of the mnemonic used.

In the real-address mode, if the ESP or SP register is 1, 3, or 5 when PUSHA/PUSHAD executes: an #SS exception is generated but not delivered (the stack error reported prevents #SS delivery). Next, the processor generates a #DF exception and enters a shutdown state as described in the #DF discussion in Chapter 7 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

```
IF 64-bit Mode
    THEN #UD
FI;
IF OperandSize = 32 (* PUSHAD instruction *)
    THEN
        Temp := (ESP);
        Push(EAX);
        Push(ECX);
        Push(EDX);
        Push(EBX);
        Push(Temp);
        Push(EBP);
        Push(ESI);
        Push(EDI);
    ELSE (* OperandSize = 16, PUSHA instruction *)
        Temp := (SP);
        Push(AX);
        Push(CX);
        Push(DX);
        Push(BX);
        Push(Temp);
```

```
        Push(BP);
        Push(SI);
        Push(DI);
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#SS(0)	If the starting or ending stack address is outside the stack segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If the ESP or SP register contains 7, 9, 11, 13, or 15.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If the ESP or SP register contains 7, 9, 11, 13, or 15.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If in 64-bit mode.

PUSHF/PUSHFD/PUSHFQ—Push EFLAGS Register Onto the Stack

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
9C	PUSHF	ZO	Valid	Valid	Push lower 16 bits of EFLAGS.
9C	PUSHFD	ZO	N.E.	Valid	Push EFLAGS.
9C	PUSHFQ	ZO	Valid	N.E.	Push RFLAGS.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Decrements the stack pointer by 4 (if the current operand-size attribute is 32) and pushes the entire contents of the EFLAGS register onto the stack, or decrements the stack pointer by 2 (if the operand-size attribute is 16) and pushes the lower 16 bits of the EFLAGS register (that is, the FLAGS register) onto the stack. These instructions reverse the operation of the POPF/POPFD instructions.

When copying the entire EFLAGS register to the stack, the VM and RF flags (bits 16 and 17) are not copied; instead, the values for these flags are cleared in the EFLAGS image stored on the stack. See Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for more information about the EFLAGS register.

The PUSHF (push flags) and PUSHFD (push flags double) mnemonics reference the same opcode. The PUSHF instruction is intended for use when the operand-size attribute is 16 and the PUSHFD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when PUSHF is used and to 32 when PUSHFD is used. Others may treat these mnemonics as synonyms (PUSHF/PUSHFD) and use the current setting of the operand-size attribute to determine the size of values to be pushed from the stack, regardless of the mnemonic used.

In 64-bit mode, the instruction's default operation is to decrement the stack pointer (RSP) by 8 and pushes RFLAGS on the stack. 16-bit operation is supported using the operand size override prefix 66H. 32-bit operand size cannot be encoded in this mode. When copying RFLAGS to the stack, the VM and RF flags (bits 16 and 17) are not copied; instead, values for these flags are cleared in the RFLAGS image stored on the stack.

When operating in virtual-8086 mode (EFLAGS.VM = 1) without the virtual-8086 mode extensions (CR4.VME = 0), the PUSHF/PUSHFD instructions can be used only if IOPL = 3; otherwise, a general-protection exception (#GP) occurs. If the virtual-8086 mode extensions are enabled (CR4.VME = 1), PUSHF (but not PUSHFD) can be executed in virtual-8086 mode with IOPL < 3.

(The protected-mode virtual-interrupt feature — enabled by setting CR4.PVI — affects the CLI and STI instructions in the same manner as the virtual-8086 mode extensions. PUSHF, however, is not affected by CR4.PVI.)

In the real-address mode, if the ESP or SP register is 1 when PUSHF/PUSHFD instruction executes: an #SS exception is generated but not delivered (the stack error reported prevents #SS delivery). Next, the processor generates a #DF exception and enters a shutdown state as described in the #DF discussion in Chapter 7 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

Operation

```
IF (PE = 0) or (PE = 1 and ((VM = 0) or (VM = 1 and IOPL = 3)))
(* Real-Address Mode, Protected mode, or Virtual-8086 mode with IOPL equal to 3 *)
    THEN
        IF OperandSize = 32
            THEN
                push (EFLAGS AND 00FCFFFFH);
                (* VM and RF bits are cleared in image stored on the stack *)
            ELSE
                push (EFLAGS); (* Lower 16 bits only *)
```

```
        FI;
    ELSE IF 64-bit MODE (* In 64-bit Mode *)
        IF OperandSize = 64
            THEN
                push (RFLAGS AND 00000000_00FCFFFFH);
                (* VM and RF bits are cleared in image stored on the stack; *)
            ELSE
                push (EFLAGS); (* Lower 16 bits only *)
        FI;
    ELSE (* In Virtual-8086 Mode with IOPL less than 3 *)
        IF (CR4.VME = 0) OR (OperandSize = 32)
            THEN #GP(0); (* Trap to virtual-8086 monitor *)
            ELSE
                tempFLAGS = EFLAGS[15:0];
                tempFLAGS[9] = tempFLAGS[19];       (* VIF replaces IF *)
                tempFlags[13:12] = 3;  (* IOPL is set to 3 in image stored on the stack *)
                push (tempFLAGS);
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#SS(0)	If the new value of the ESP register is outside the stack segment boundary.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while CPL = 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If the I/O privilege level is less than 3.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while CPL = 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

PXOR—Logical Exclusive OR

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EF /r[1] PXOR mm, mm/m64	A	V/V	MMX	Bitwise XOR of mm/m64 and mm.
66 0F EF /r PXOR xmm1, xmm2/m128	A	V/V	SSE2	Bitwise XOR of xmm2/m128 and xmm1.
VEX.128.66.0F.WIG EF /r VPXOR xmm1, xmm2, xmm3/m128	B	V/V	AVX	Bitwise XOR of xmm3/m128 and xmm2.
VEX.256.66.0F.WIG EF /r VPXOR ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Bitwise XOR of ymm3/m256 and ymm2.
EVEX.128.66.0F.W0 EF /r VPXORD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise XOR of packed doubleword integers in xmm2 and xmm3/m128 using writemask k1.
EVEX.256.66.0F.W0 EF /r VPXORD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise XOR of packed doubleword integers in ymm2 and ymm3/m256 using writemask k1.
EVEX.512.66.0F.W0 EF /r VPXORD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Bitwise XOR of packed doubleword integers in zmm2 and zmm3/m512/m32bcst using writemask k1.
EVEX.128.66.0F.W1 EF /r VPXORQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise XOR of packed quadword integers in xmm2 and xmm3/m128 using writemask k1.
EVEX.256.66.0F.W1 EF /r VPXORQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise XOR of packed quadword integers in ymm2 and ymm3/m256 using writemask k1.
EVEX.512.66.0F.W1 EF /r VPXORQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Bitwise XOR of packed quadword integers in zmm2 and zmm3/m512/m64bcst using writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 25.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical exclusive-OR (XOR) operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. Each bit of the result is 1 if the corresponding bits of the two operands are different; each bit is 0 if the corresponding bits of the operands are the same.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding register destination are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with write-mask k1.

Operation

PXOR (64-bit Operand)

```
DEST := DEST XOR SRC
```

PXOR (128-bit Legacy SSE Version)

```
DEST := DEST XOR SRC
DEST[MAXVL-1:128] (Unmodified)
```

VPXOR (VEX.128 Encoded Version)

```
DEST := SRC1 XOR SRC2
DEST[MAXVL-1:128] := 0
```

VPXOR (VEX.256 Encoded Version)

```
DEST := SRC1 XOR SRC2
DEST[MAXVL-1:256] := 0
```

VPXORD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := SRC1[i+31:i] BITWISE XOR SRC2[31:0]
                ELSE DEST[i+31:i] := SRC1[i+31:i] BITWISE XOR SRC2[i+31:i]
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPXORQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := SRC1[i+63:i] BITWISE XOR SRC2[63:0]
                ELSE DEST[i+63:i] := SRC1[i+63:i] BITWISE XOR SRC2[i+63:i]
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63:0] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPXORD __m512i _mm512_xor_epi32(__m512i a, __m512i b)
VPXORD __m512i _mm512_mask_xor_epi32(__m512i s, __mmask16 m, __m512i a, __m512i b)
VPXORD __m512i _mm512_maskz_xor_epi32(__mmask16 m, __m512i a, __m512i b)
VPXORD __m256i _mm256_xor_epi32(__m256i a, __m256i b)
VPXORD __m256i _mm256_mask_xor_epi32(__m256i s, __mmask8 m, __m256i a, __m256i b)
VPXORD __m256i _mm256_maskz_xor_epi32(__mmask8 m, __m256i a, __m256i b)
VPXORD __m128i _mm_xor_epi32(__m128i a, __m128i b)
VPXORD __m128i _mm_mask_xor_epi32(__m128i s, __mmask8 m, __m128i a, __m128i b)
VPXORD __m128i _mm_maskz_xor_epi32(__mmask16 m, __m128i a, __m128i b)
VPXORQ __m512i _mm512_xor_epi64(__m512i a, __m512i b);
VPXORQ __m512i _mm512_mask_xor_epi64(__m512i s, __mmask8 m, __m512i a, __m512i b);
VPXORQ __m512i _mm512_maskz_xor_epi64(__mmask8 m, __m512i a, __m512i b);
VPXORQ __m256i _mm256_xor_epi64(__m256i a, __m256i b);
VPXORQ __m256i _mm256_mask_xor_epi64(__m256i s, __mmask8 m, __m256i a, __m256i b);
VPXORQ __m256i _mm256_maskz_xor_epi64(__mmask8 m, __m256i a, __m256i b);
VPXORQ __m128i _mm_xor_epi64(__m128i a, __m128i b);
VPXORQ __m128i _mm_mask_xor_epi64(__m128i s, __mmask8 m, __m128i a, __m128i b);
VPXORQ __m128i _mm_maskz_xor_epi64(__mmask8 m, __m128i a, __m128i b);
PXOR:__m64 _mm_xor_si64 (__m64 m1, __m64 m2)
(V)PXOR:__m128i _mm_xor_si128 (__m128i a, __m128i b)
VPXOR:__m256i _mm256_xor_si256 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

RCL/RCR/ROL/ROR—Rotate

Opcode[1]	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
D0 /2	RCL r/m8[2], 1	M1	Valid	Valid	Rotate 9 bits (CF, r/m8) left once.
D2 /2	RCL r/m8[2], CL	MC	Valid	Valid	Rotate 9 bits (CF, r/m8) left CL times.
C0 /2 ib	RCL r/m8[2], imm8	MI	Valid	Valid	Rotate 9 bits (CF, r/m8) left imm8 times.
D1 /2	RCL r/m16, 1	M1	Valid	Valid	Rotate 17 bits (CF, r/m16) left once.
D3 /2	RCL r/m16, CL	MC	Valid	Valid	Rotate 17 bits (CF, r/m16) left CL times.
C1 /2 ib	RCL r/m16, imm8	MI	Valid	Valid	Rotate 17 bits (CF, r/m16) left imm8 times.
D1 /2	RCL r/m32, 1	M1	Valid	Valid	Rotate 33 bits (CF, r/m32) left once.
REX.W + D1 /2	RCL r/m64, 1	M1	Valid	N.E.	Rotate 65 bits (CF, r/m64) left once. Uses a 6 bit count.
D3 /2	RCL r/m32, CL	MC	Valid	Valid	Rotate 33 bits (CF, r/m32) left CL times.
REX.W + D3 /2	RCL r/m64, CL	MC	Valid	N.E.	Rotate 65 bits (CF, r/m64) left CL times. Uses a 6 bit count.
C1 /2 ib	RCL r/m32, imm8	MI	Valid	Valid	Rotate 33 bits (CF, r/m32) left imm8 times.
REX.W + C1 /2 ib	RCL r/m64, imm8	MI	Valid	N.E.	Rotate 65 bits (CF, r/m64) left imm8 times. Uses a 6 bit count.
D0 /3	RCR r/m8[2], 1	M1	Valid	Valid	Rotate 9 bits (CF, r/m8) right once.
D2 /3	RCR r/m8[2], CL	MC	Valid	Valid	Rotate 9 bits (CF, r/m8) right CL times.
C0 /3 ib	RCR r/m8[2], imm8	MI	Valid	Valid	Rotate 9 bits (CF, r/m8) right imm8 times.
D1 /3	RCR r/m16, 1	M1	Valid	Valid	Rotate 17 bits (CF, r/m16) right once.
D3 /3	RCR r/m16, CL	MC	Valid	Valid	Rotate 17 bits (CF, r/m16) right CL times.
C1 /3 ib	RCR r/m16, imm8	MI	Valid	Valid	Rotate 17 bits (CF, r/m16) right imm8 times.
D1 /3	RCR r/m32, 1	M1	Valid	Valid	Rotate 33 bits (CF, r/m32) right once. Uses a 6 bit count.
REX.W + D1 /3	RCR r/m64, 1	M1	Valid	N.E.	Rotate 65 bits (CF, r/m64) right once. Uses a 6 bit count.
D3 /3	RCR r/m32, CL	MC	Valid	Valid	Rotate 33 bits (CF, r/m32) right CL times.
REX.W + D3 /3	RCR r/m64, CL	MC	Valid	N.E.	Rotate 65 bits (CF, r/m64) right CL times. Uses a 6 bit count.
C1 /3 ib	RCR r/m32, imm8	MI	Valid	Valid	Rotate 33 bits (CF, r/m32) right imm8 times.
REX.W + C1 /3 ib	RCR r/m64, imm8	MI	Valid	N.E.	Rotate 65 bits (CF, r/m64) right imm8 times. Uses a 6 bit count.
D0 /0	ROL r/m8,[2] 1	M1	Valid	Valid	Rotate 8 bits r/m8 left once.
D2 /0	ROL r/m8[2], CL	MC	Valid	Valid	Rotate 8 bits r/m8 left CL times.
C0 /0 ib	ROL r/m8[2], imm8	MI	Valid	Valid	Rotate 8 bits r/m8 left imm8 times.
D1 /0	ROL r/m16, 1	M1	Valid	Valid	Rotate 16 bits r/m16 left once.
D3 /0	ROL r/m16, CL	MC	Valid	Valid	Rotate 16 bits r/m16 left CL times.
C1 /0 ib	ROL r/m16, imm8	MI	Valid	Valid	Rotate 16 bits r/m16 left imm8 times.
D1 /0	ROL r/m32, 1	M1	Valid	Valid	Rotate 32 bits r/m32 left once.
REX.W + D1 /0	ROL r/m64, 1	M1	Valid	N.E.	Rotate 64 bits r/m64 left once. Uses a 6 bit count.
D3 /0	ROL r/m32, CL	MC	Valid	Valid	Rotate 32 bits r/m32 left CL times.

Opcode[1]	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
REX.W + D3 /0	ROL r/m64, CL	MC	Valid	N.E.	Rotate 64 bits r/m64 left CL times. Uses a 6 bit count.
C1 /0 ib	ROL r/m32, imm8	MI	Valid	Valid	Rotate 32 bits r/m32 left imm8 times.
REX.W + C1 /0 ib	ROL r/m64, imm8	MI	Valid	N.E.	Rotate 64 bits r/m64 left imm8 times. Uses a 6 bit count.
D0 /1	ROR r/m8[2], 1	M1	Valid	Valid	Rotate 8 bits r/m8 right once.
D2 /1	ROR r/m8[2], CL	MC	Valid	Valid	Rotate 8 bits r/m8 right CL times.
C0 /1 ib	ROR r/m8[2], imm8	MI	Valid	Valid	Rotate 8 bits r/m16 right imm8 times.
D1 /1	ROR r/m16, 1	M1	Valid	Valid	Rotate 16 bits r/m16 right once.
D3 /1	ROR r/m16, CL	MC	Valid	Valid	Rotate 16 bits r/m16 right CL times.
C1 /1 ib	ROR r/m16, imm8	MI	Valid	Valid	Rotate 16 bits r/m16 right imm8 times.
D1 /1	ROR r/m32, 1	M1	Valid	Valid	Rotate 32 bits r/m32 right once.
REX.W + D1 /1	ROR r/m64, 1	M1	Valid	N.E.	Rotate 64 bits r/m64 right once. Uses a 6 bit count.
D3 /1	ROR r/m32, CL	MC	Valid	Valid	Rotate 32 bits r/m32 right CL times.
REX.W + D3 /1	ROR r/m64, CL	MC	Valid	N.E.	Rotate 64 bits r/m64 right CL times. Uses a 6 bit count.
C1 /1 ib	ROR r/m32, imm8	MI	Valid	Valid	Rotate 32 bits r/m32 right imm8 times.
REX.W + C1 /1 ib	ROR r/m64, imm8	MI	Valid	N.E.	Rotate 64 bits r/m64 right imm8 times. Uses a 6 bit count.

NOTES:

1. See the IA-32 Architecture Compatibility section below.
2. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M1	ModRM:r/m (w)	1	N/A	N/A
MC	ModRM:r/m (w)	CL	N/A	N/A
MI	ModRM:r/m (w)	imm8	N/A	N/A

Description

Shifts (rotates) the bits of the first operand (destination operand) the number of bit positions specified in the second operand (count operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the count operand is an unsigned integer that can be an immediate or a value in the CL register. The count is masked to 5 bits (or 6 bits if in 64-bit mode and REX.W = 1).

The rotate left (ROL) and rotate through carry left (RCL) instructions shift all the bits toward more-significant bit positions, except for the most-significant bit, which is rotated to the least-significant bit location. The rotate right (ROR) and rotate through carry right (RCR) instructions shift all the bits toward less significant bit positions, except for the least-significant bit, which is rotated to the most-significant bit location.

The RCL and RCR instructions include the CF flag in the rotation. The RCL instruction shifts the CF flag into the least-significant bit and shifts the most-significant bit into the CF flag. The RCR instruction shifts the CF flag into the most-significant bit and shifts the least-significant bit into the CF flag. For the ROL and ROR instructions, the original value of the CF flag is not a part of the result, but the CF flag receives a copy of the bit that was shifted from one end to the other.

The OF flag is defined only for the 1-bit rotates; it is undefined in all other cases (except RCL and RCR instructions only: a zero-bit rotate does nothing, that is affects no flags). For left rotates, the OF flag is set to the exclusive OR of the CF bit (after the rotate) and the most-significant bit of the result. For right rotates, the OF flag is set to the exclusive OR of the two most-significant bits of the result.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Use of REX.W promotes the first operand to 64 bits and causes the count operand to become a 6-bit counter.

IA-32 Architecture Compatibility

The 8086 does not mask the rotation count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the rotation count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

Operation

```
(* RCL and RCR Instructions *)
SIZE := OperandSize;
CASE (determine count) OF
    SIZE := 8:      tempCOUNT := (COUNT AND 1FH) MOD 9;
    SIZE := 16:     tempCOUNT := (COUNT AND 1FH) MOD 17;
    SIZE := 32:     tempCOUNT := COUNT AND 1FH;
    SIZE := 64:     tempCOUNT := COUNT AND 3FH;
ESAC;
IF OperandSize = 64
    THEN COUNTMASK = 3FH;
    ELSE COUNTMASK = 1FH;
FI;

(* RCL Instruction Operation *)
tempDEST := DEST;
WHILE (tempCOUNT ≠ 0)
    DO
        tempCF := MSB(tempDEST);
        tempDEST := (tempDEST * 2) + CF;
        CF := tempCF;
        tempCOUNT := tempCOUNT – 1;
    OD;
ELIHW;
IF (COUNT & COUNTMASK) = 1
    THEN OF := MSB(tempDEST) XOR CF;
    ELSE OF is undefined;
FI;
DEST := tempDEST;
```

```
(* RCR Instruction Operation *)
tempDEST := DEST;
IF (COUNT & COUNTMASK) = 1
    THEN OF := MSB(tempDEST) XOR CF;
    ELSE OF is undefined;
FI;
WHILE (tempCOUNT ≠ 0)
    DO
        tempCF := LSB(SRC);
        tempDEST := (tempDEST / 2) + (CF * 2^SIZE);
        CF := tempCF;
        tempCOUNT := tempCOUNT – 1;
    OD;
DEST := tempDEST;

(* ROL Instruction Operation *)
tempCOUNT := (COUNT & COUNTMASK) MOD SIZE
tempDEST := DEST;
WHILE (tempCOUNT ≠ 0)
    DO
        tempCF := MSB(tempDEST);
        tempDEST := (tempDEST ∗ 2) + tempCF;
        tempCOUNT := tempCOUNT – 1;
    OD;
ELIHW;
IF (COUNT & COUNTMASK) ≠ 0
    THEN CF := LSB(tempDEST);
FI;
IF (COUNT & COUNTMASK) = 1
    THEN OF := MSB(tempDEST) XOR CF;
    ELSE OF is undefined;
FI;
DEST := tempDEST;

(* ROR Instruction Operation *)
tempCOUNT := (COUNT & COUNTMASK) MOD SIZE
tempDEST := DEST;
WHILE (tempCOUNT ≠ 0)
    DO
        tempCF := LSB(SRC);
        tempDEST := (tempDEST / 2) + (tempCF ∗ 2^SIZE);
        tempCOUNT := tempCOUNT – 1;
    OD;
ELIHW;
IF (COUNT & COUNTMASK) ≠ 0
    THEN CF := MSB(tempDEST);
FI;
IF (COUNT & COUNTMASK) = 1
    THEN OF := MSB(tempDEST) XOR MSB – 1(tempDEST);
    ELSE OF is undefined;
FI;
DEST := tempDEST;
```

Flags Affected

For RCL and RCR instructions, a zero-bit rotate does nothing, i.e., affects no flags. For ROL and ROR instructions, if the masked count is 0, the flags are not affected. If the masked count is 1, then the OF flag is affected, otherwise (masked count is greater than 1) the OF flag is undefined.

For all instructions, the CF flag is affected when the masked count is non-zero. The SF, ZF, AF, and PF flags are always unaffected.

Protected Mode Exceptions

#GP(0)	If the source operand is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the source operand is located in a nonwritable segment.
	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

RCPPS—Compute Reciprocals of Packed Single Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 53 /r RCPPS xmm1, xmm2/m128	RM	V/V	SSE	Computes the approximate reciprocals of the packed single precision floating-point values in xmm2/m128 and stores the results in xmm1.
VEX.128.0F.WIG 53 /r VRCPPS xmm1, xmm2/m128	RM	V/V	AVX	Computes the approximate reciprocals of packed single precision values in xmm2/mem and stores the results in xmm1.
VEX.256.0F.WIG 53 /r VRCPPS ymm1, ymm2/m256	RM	V/V	AVX	Computes the approximate reciprocals of packed single precision values in ymm2/mem and stores the results in ymm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Performs a SIMD computation of the approximate reciprocals of the four packed single precision floating-point values in the source operand (second operand) stores the packed single precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD single precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RCPPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). Tiny results (see Section 4.9.1.5, "Numeric Underflow Exception (#U)" in Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1) are always flushed to 0.0, with the sign of the operand. (Input values greater than or equal to $|1.11111111110100000000000B*2^{125}|$ are guaranteed to not produce tiny results; input values less than or equal to $|1.00000000000110000000001B*2^{126}|$ are guaranteed to produce tiny results, which are in turn flushed to 0.0; and input values in between this range may or may not produce tiny results, depending on the implementation.) When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

RCPPS (128-bit Legacy SSE Version)

```
DEST[31:0] := APPROXIMATE(1/SRC[31:0])
DEST[63:32] := APPROXIMATE(1/SRC[63:32])
DEST[95:64] := APPROXIMATE(1/SRC[95:64])
DEST[127:96] := APPROXIMATE(1/SRC[127:96])
DEST[MAXVL-1:128] (Unmodified)
```

VRCPPS (VEX.128 Encoded Version)

```
DEST[31:0] := APPROXIMATE(1/SRC[31:0])
DEST[63:32] := APPROXIMATE(1/SRC[63:32])
DEST[95:64] := APPROXIMATE(1/SRC[95:64])
DEST[127:96] := APPROXIMATE(1/SRC[127:96])
DEST[MAXVL-1:128] := 0
```

VRCPPS (VEX.256 Encoded Version)

```
DEST[31:0] := APPROXIMATE(1/SRC[31:0])
DEST[63:32] := APPROXIMATE(1/SRC[63:32])
DEST[95:64] := APPROXIMATE(1/SRC[95:64])
DEST[127:96] := APPROXIMATE(1/SRC[127:96])
DEST[159:128] := APPROXIMATE(1/SRC[159:128])
DEST[191:160] := APPROXIMATE(1/SRC[191:160])
DEST[223:192] := APPROXIMATE(1/SRC[223:192])
DEST[255:224] := APPROXIMATE(1/SRC[255:224])
```

Intel C/C++ Compiler Intrinsic Equivalent

RCCPS __m128 _mm_rcp_ps(__m128 a)
RCPPS __m256 _mm256_rcp_ps (__m256 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions," additionally:

#UD	If VEX.vvvv ≠ 1111B.

RCPSS—Compute Reciprocal of Scalar Single Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 53 /r RCPSS xmm1, xmm2/m32	RM	V/V	SSE	Computes the approximate reciprocal of the scalar single precision floating-point value in xmm2/m32 and stores the result in xmm1.
VEX.LIG.F3.0F.WIG 53 /r VRCPSS xmm1, xmm2, xmm3/m32	RVM	V/V	AVX	Computes the approximate reciprocal of the scalar single precision floating-point value in xmm3/m32 and stores the result in xmm1. Also, upper single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Computes of an approximate reciprocal of the low single precision floating-point value in the source operand (second operand) and stores the single precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a scalar single precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RCPSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). Tiny results (see Section 4.9.1.5, "Numeric Underflow Exception (#U)" in Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1) are always flushed to 0.0, with the sign of the operand. (Input values greater than or equal to $|1.11111111110100000000000B*2^{125}|$ are guaranteed to not produce tiny results; input values less than or equal to $|1.00000000000110000000001B*2^{126}|$ are guaranteed to produce tiny results, which are in turn flushed to 0.0; and input values in between this range may or may not produce tiny results, depending on the implementation.) When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

Operation

RCPSS (128-bit Legacy SSE Version)

```
DEST[31:0] := APPROXIMATE(1/SRC[31:0])
DEST[MAXVL-1:32] (Unmodified)
```

VRCPSS (VEX.128 Encoded Version)
DEST[31:0] := APPROXIMATE(1/SRC2[31:0])
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent

RCPSS __m128 _mm_rcp_ss(__m128 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-22, "Type 5 Class Exception Conditions."

RDFSBASE/RDGSBASE—Read FS/GS Segment Base

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F AE /0 RDFSBASE r32	M	V/I	FSGSBASE	Load the 32-bit destination register with the FS base address.
F3 REX.W 0F AE /0 RDFSBASE r64	M	V/I	FSGSBASE	Load the 64-bit destination register with the FS base address.
F3 0F AE /1 RDGSBASE r32	M	V/I	FSGSBASE	Load the 32-bit destination register with the GS base address.
F3 REX.W 0F AE /1 RDGSBASE r64	M	V/I	FSGSBASE	Load the 64-bit destination register with the GS base address.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	N/A	N/A	N/A

Description

Loads the general-purpose register indicated by the ModR/M:r/m field with the FS or GS segment base address.

The destination operand may be either a 32-bit or a 64-bit general-purpose register. The REX.W prefix indicates the operand size is 64 bits. If no REX.W prefix is used, the operand size is 32 bits; the upper 32 bits of the source base address (for FS or GS) are ignored and upper 32 bits of the destination register are cleared.

This instruction is supported only in 64-bit mode.

Operation

```
DEST := FS/GS segment base address;
```

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

RDFSBASE unsigned int _readfsbase_u32(void);
RDFSBASE unsigned __int64 _readfsbase_u64(void);
RDGSBASE unsigned int _readgsbase_u32(void);
RDGSBASE unsigned __int64 _readgsbase_u64(void);

Protected Mode Exceptions

#UD	The RDFSBASE and RDGSBASE instructions are not recognized in protected mode.

Real-Address Mode Exceptions

#UD	The RDFSBASE and RDGSBASE instructions are not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The RDFSBASE and RDGSBASE instructions are not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	The RDFSBASE and RDGSBASE instructions are not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used. If CR4.FSGSBASE[bit 16] = 0. If CPUID.07H.00H:EBX.FSGSBASE[0] = 0.

RDMSR—Read From Model Specific Register

Opcode[1]/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
0F 32 RDMSR	ZO	Valid		Read MSR specified by ECX into EDX:EAX.
VEX.128.F2.MAP7:W0.F6 11:000:bbb RDMSR r64, imm32	MI	V/N.E.	MSR_IMM	Load into register bbb the value of the MSR with address in the 32-bit immediate.

NOTES:

1. See the IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A
MI	ModRM:r/m (r)	imm32	N/A	N/A

Description

Reads the contents of a 64-bit model specific register (MSR). RDMSR has an **implicit** form and an **immediate** form.

The implicit form reads the contents of a 64-bit model specific register (MSR) specified in the ECX register into registers EDX:EAX. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.) If fewer than 64 bits are implemented in the MSR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

The immediate form reads the contents of the MSR specified by operand 2 into operand 1. Operand 2 is an immediate, while operand 1 is a general-purpose register. The immediate form can be used only in 64-bit mode; otherwise, a invalid-opcode exception (#UD) will be generated. The immediate form may provide better performance than the implicit form.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) will be generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception.

The MSRs control functions for testability, execution tracing, performance-monitoring, and machine check errors. Chapter 2, "Model-Specific Registers (MSRs)" of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4, lists all the MSRs that can be read with this instruction and their addresses. Note that each processor family has its own set of MSRs.

The CPUID instruction should be used to determine whether MSRs are supported (CPUID.01H:EDX[5] = 1) before using this instruction. The immediate form is supported only in CPUID.07H.01H:ECX.MSR_IMM[5] = 1.

IA-32 Architecture Compatibility

The MSRs and the ability to read them with the RDMSR instruction were introduced into the IA-32 Architecture with the Pentium processor. Execution of this instruction by an IA-32 processor earlier than the Pentium processor results in an invalid opcode exception #UD.

Operation

```
IF implicit form
    THEN
        EDX:EAX := MSR[ECX];
```

```
    ELSE (* immediate form *)
        DEST := MSR[SRC];
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If the value in ECX specifies a reserved or unimplemented MSR address.
#UD	If the LOCK prefix is used.
	If the immediate form is used.

Real-Address Mode Exceptions

#GP	If the value in ECX specifies a reserved or unimplemented MSR address.
#UD	If the LOCK prefix is used.
	If the immediate form is used.

Virtual-8086 Mode Exceptions

#GP(0)	The RDMSR instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If the value in ECX specifies a reserved or unimplemented MSR address.
#UD	If the LOCK prefix is used.
	If the immediate form is used and CPUID.07H.01H:ECX.MSR_IMM[5] = 0.

RDMSRLIST—Read List of Model Specific Registers

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 01 C6 RDMSRLIST	ZO	V/N.E.	MSRLIST	Read the requested list of MSRs, and store the read values to memory.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

This instruction reads a software-provided list of up to 64 MSRs and stores their values in memory.

RDMSRLIST takes three implied input operands:

- RSI: Linear address of a table of MSR addresses (8 bytes per address)[1].
- RDI: Linear address of a table into which MSR data is stored (8 bytes per MSR).
- RCX: 64-bit bitmask of valid bits for the MSRs. Bit 0 is the valid bit for entry 0 in each table, etc.

For each RCX bit [n] from 0 to 63, if RCX[n] is 1, RDMSRLIST will read the MSR specified at entry [n] in the RSI-based table and write it out to memory at the entry [n] in the RDI-based table.

This implies a maximum of 64 MSRs that can be processed by this instruction. The processor will clear RCX[n] after it finishes handling that MSR. Similar to repeated string operations, RDMSRLIST supports partial completion for interrupts, exceptions, and traps. In these situations, the RIP register saved will point to the RDMSRLIST instruction while the RCX register will have cleared bits corresponding to all completed iterations.

This instruction must be executed at privilege level 0; otherwise, a general protection exception #GP(0) is generated. This instruction performs MSR-specific checks in the same manner as RDMSR.

Although RDMSRLIST accesses the entries in the two tables in order, the actual reads of the MSRs may be performed out of order: for table entries m < n, the processor may read the MSR for entry n before reading the MSR for entry m. (This may be true also for a sequence of executions of RDMSR.) Ordering is guaranteed if the address of the IA32_BARRIER MSR (2FH) appears in the table of MSR addresses. Specifically, if IA32_BARRIER appears at entry m, then the MSR read for any entry n with n > m will not occur until (1) all instructions prior to RDMSRLIST have completed locally; and (2) MSRs have been read for all table entries before entry m.

The processor is allowed (but not required) to "load ahead" in the list. For example, it may cause a page fault for an access to a table entry after the n^{th}, despite the processor having read only n MSRs.[2]

Operation

```
DO WHILE RCX != 0
    MSR_index := position of least significant bit set in RCX;
    Load MSR_address_table_entry from 8 bytes at the linear address RSI + (MSR_index * 8);
    IF MSR_address_table_entry[63:32] != 0 THEN #GP(0); FI;
    MSR_address := MSR_address_table_entry[31:0];
    IF RDMSR of the MSR with address MSR_address would #GP THEN #GP(0); FI;
    Store the value of the MSR with address MSR_address into 8 bytes at the linear address RDI + (MSR_index * 8);
    RCX[MSR_index] := 0;
    Allow delivery of any pending interrupts or traps;
OD;
```

1. Since MSR addresses are only 32-bits wide, bits 63:32 of each MSR address table entry is reserved.
2. For example, the processor may take a page fault due to a linear address for the 10th entry in the MSR address table despite only having completed the MSR reads up to entry 5.

Flags Affected

None.

Protected Mode Exceptions

#UD	The RDMSRLIST instruction is not recognized in protected mode.

Real-Address Mode Exceptions

#UD	The RDMSRLIST instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The RDMSRLIST instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	The RDMSRLIST instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If RSI [2:0] ≠ 0, RDI [2:0] ≠ 0, or bits 63:32 of an MSR-address table entry are not all zero.
	If an execution of RDMSR from a specified MSR would generate a general protection exception #GP(0).
	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.
	If CPUID.07H.01H:EAX.MSRLIST[27] = 0.

RDPID—Read Processor ID

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F C7 /7 RDPID r32	R	N.E./V	RDPID	Read IA32_TSC_AUX into r32.
F3 0F C7 /7 RDPID r64	R	V/N.E.	RDPID	Read IA32_TSC_AUX into r64.

Instruction Operand Encoding[1]

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
R	ModRM:r/m (w)	N/A	N/A	N/A

Description

Reads the value of the IA32_TSC_AUX MSR (address C0000103H) into the destination register. The value of CS.D and operand-size prefixes (66H and REX.W) do not affect the behavior of the RDPID instruction.

Operation

```
DEST := IA32_TSC_AUX
```

Flags Affected

None.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.07H.00H:ECX.RDPID[22] = 0.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

1. ModRM.MOD = 011B required

RDPKRU—Read Protection Key Rights for User Pages

Opcode*	Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 EE	RDPKRU	ZO	V/V	OSPKE	Reads PKRU into EAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Reads the value of PKRU into EAX and clears EDX. ECX must be 0 when RDPKRU is executed; otherwise, a general-protection exception (#GP) occurs.

RDPKRU can be executed only if CR4.PKE = 1; otherwise, an invalid-opcode exception (#UD) occurs. Software can discover the value of CR4.PKE by examining CPUID.07H.00H:ECX.OSPKE[4].

On processors that support the Intel 64 Architecture, the high-order 32-bits of RCX are ignored and the high-order 32-bits of RDX and RAX are cleared.

Operation

```
IF (ECX = 0)
    THEN
        EAX := PKRU;
        EDX := 0;
    ELSE #GP(0);
FI;
```

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

RDPKRU uint32_t _rdpkru_u32(void);

Protected Mode Exceptions

#GP(0)	If ECX ≠ 0.
#UD	If the LOCK prefix is used.
	If CR4.PKE = 0.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

RDPMC—Read Performance-Monitoring Counters

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 33	RDPMC	ZO	Valid	Valid	Read performance-monitoring counter specified by ECX into EDX:EAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Reads the contents of the performance monitoring counter (PMC) specified in ECX register into registers EDX:EAX. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The EDX register is loaded with the high-order 32 bits of the PMC and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.) If fewer than 64 bits are implemented in the PMC being read, unimplemented bits returned to EDX:EAX will have value zero.

The width of PMCs on processors supporting architectural performance monitoring (CPUID.0AH:EAX[7:0] ≠ 0) are reported by CPUID.0AH:EAX[23:16]. On processors that do not support architectural performance monitoring (CPUID.0AH:EAX[7:0]=0), the width of general-purpose performance PMCs is 40 bits, while the widths of special-purpose PMCs are implementation specific.

Use of ECX to specify a PMC depends on whether the processor supports architectural performance monitoring:

- If the processor does not support architectural performance monitoring (CPUID.0AH:EAX[7:0]=0), ECX[30:0] specifies the index of the PMC to be read. Setting ECX[31] selects "fast" read mode if supported. In this mode, RDPMC returns bits 31:0 of the PMC in EAX while clearing EDX to zero.
- If the processor does support architectural performance monitoring (CPUID.0AH:EAX[7:0] ≠ 0), ECX[31:16] specifies type of PMC while ECX[15:0] specifies the index of the PMC to be read within that type. The following PMC types are currently defined:
  - General-purpose counters use type 0. To read IA32_PMCx, one of the following must hold for the index x:
    - It is less than the value enumerated by CPUID.0AH:EAX[15:8]; or
    - It is at most 31 and the value enumerated by CPUID.23H.01H:EAX[x] is 1.
  - Fixed-function counters use type 4000H. To read IA32_FIXED_CTRx, one of the following must hold for the index x:
    - It is less than the value enumerated by CPUID.0AH:EDX[4:0];
    - It is at most 31 and the value enumerated by CPUID.0AH:ECX[x] is 1; or
    - It is at most 31 and the value enumerated by CPUID.23H.01H:EBX[x] is 1.
  - Performance metrics use type 2000H. This type can be used only if IA32_PERF_CAPABILITIES.PERF_METRICS_AVAILABLE[bit 15]=1. For this type, the index in ECX[15:0] is implementation specific.

Specifying an unsupported PMC encoding will cause a general protection exception #GP(0). For PMC details see Chapter 21, "Last Branch Records," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

When in protected or virtual 8086 mode, the **Performance-monitoring Counters Enabled** (PCE) flag in register CR4 restricts the use of the RDPMC instruction. When the PCE flag is set, the RDPMC instruction can be executed at any privilege level; when the flag is clear, the instruction can only be executed at privilege level 0. (When in real-address mode, the RDPMC instruction is always enabled.) The PMCs can also be read with the RDMSR instruction, when executing at privilege level 0.

Processors that support performance metrics may also support clearing them on read if the IA32_PERF_CAPABILITIES.RDPMC_METRICS_CLEAR[bit 19] is set. Since the IA32_PERF_CAPABILITIES MSR

enumerates non-architectural PMU features, software should check DisplayFamily and DisplayModel to confirm that the processor supports the functionality described in the next paragraph.

When the IA32_FIXED_CTR_CTRL.METRICS_CLEAR_EN[bit 14] is set, an RDPMC instruction for PERF_METRICS (that is, when ECX=0x2000'0000) clears PERF_METRICS-related resources as well as fixed-function performance monitoring counter 3 after the read is performed. When METRICS_CLEAR_EN is clear, the RDPMC instruction only reads PERF_METRICS.

The RDPMC instruction is not a serializing instruction; that is, it does not imply that all the events caused by the preceding instructions have been completed or that events caused by subsequent instructions have not begun. If an exact event count is desired, software must insert a serializing instruction (such as the CPUID instruction) before and/or after the RDPMC instruction.

Performing back-to-back fast reads are not guaranteed to be monotonic. To guarantee monotonicity on back-to-back reads, a serializing instruction must be placed between the two RDPMC instructions.

The RDPMC instruction can execute in 16-bit addressing mode or virtual-8086 mode; however, the full contents of the ECX register are used to select the PMC, and the event count is stored in the full EAX and EDX registers. The RDPMC instruction was introduced into the IA-32 Architecture in the Pentium Pro processor and the Pentium processor with MMX technology. The earlier Pentium processors have PMCs, but they must be read with the RDMSR instruction.

Operation

```
MSCB = Most Significant Counter Bit (* Model-specific *)
IF (((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0)) and (ECX indicates a supported counter))
    THEN
        EAX := counter[31:0];
        EDX := ZeroExtend(counter[MSCB:32]);
    ELSE (* ECX is not valid or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
        #GP(0);
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.
	If an invalid performance counter index is specified.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If an invalid performance counter index is specified.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If the PCE flag in the CR4 register is clear.
	If an invalid performance counter index is specified.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0 and the PCE flag in the CR4 register is clear. If an invalid performance counter index is specified.
#UD	If the LOCK prefix is used.

RDRAND—Read Random Number

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NFx 0F C7 /6 RDRAND r16	M	V/V	RDRAND	Read a 16-bit random number and store in the destination register.
NFx 0F C7 /6 RDRAND r32	M	V/V	RDRAND	Read a 32-bit random number and store in the destination register.
NFx REX.W + 0F C7 /6 RDRAND r64	M	V/I	RDRAND	Read a 64-bit random number and store in the destination register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	N/A	N/A	N/A

Description

Loads a hardware generated random value and store it in the destination register. The size of the random value is determined by the destination register size and operating mode. The Carry Flag indicates whether a random value is available at the time the instruction is executed. CF=1 indicates that the data in the destination is valid. Otherwise CF=0 and the data in the destination operand will be returned as zeros for the specified width. All other flags are forced to 0 in either situation. Software must check the state of CF=1 for determining if a valid random value has been returned, otherwise it is expected to loop and retry execution of RDRAND (see Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, Section 7.3.17, "Random Number Generator Instructions").

This instruction is available at all privilege levels.

In 64-bit mode, the instruction's default operand size is 32 bits. Using a REX prefix in the form of REX.B permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF HW_RND_GEN.ready = 1
    THEN
        CASE of
            operand size is 64: DEST[63:0] := HW_RND_GEN.data;
            operand size is 32: DEST[31:0] := HW_RND_GEN.data;
            operand size is 16: DEST[15:0] := HW_RND_GEN.data;
        ESAC
        CF := 1;
    ELSE
        CASE of
            operand size is 64: DEST[63:0] := 0;
            operand size is 32: DEST[31:0] := 0;
            operand size is 16: DEST[15:0] := 0;
        ESAC
        CF := 0;
FI
OF, SF, ZF, AF, PF := 0;
```

Flags Affected

The CF flag is set according to the result (see the "Operation" section above). The OF, SF, ZF, AF, and PF flags are set to 0.

Intel C/C++ Compiler Intrinsic Equivalent

RDRAND int _rdrand16_step(unsigned short *);
RDRAND int _rdrand32_step(unsigned int *);
RDRAND int _rdrand64_step(unsigned __int64 *);

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.01H:ECX.RDRAND[30] = 0.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

RDSEED—Read Random SEED

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NFx 0F C7 /7 RDSEED r16	M	V/V	RDSEED	Read a 16-bit NIST SP800-90B & C compliant random value and store in the destination register.
NFx 0F C7 /7 RDSEED r32	M	V/V	RDSEED	Read a 32-bit NIST SP800-90B & C compliant random value and store in the destination register.
NFx REX.W + 0F C7 /7 RDSEED r64	M	V/I	RDSEED	Read a 64-bit NIST SP800-90B & C compliant random value and store in the destination register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	N/A	N/A	N/A

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 operand 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
            operand size is 64: DEST[63:0] := HW_NRND_GEN.data;
            operand size is 32: DEST[31:0] := HW_NRND_GEN.data;
            operand size is 16: DEST[15:0] := HW_NRND_GEN.data;
        ESAC;
        CF := 1;
    ELSE
        CASE of
            operand size is 64: DEST[63:0] := 0;
            operand size is 32: DEST[31:0] := 0;
            operand size is 16: DEST[15:0] := 0;
        ESAC;
        CF := 0;
FI;
OF, SF, ZF, AF, PF := 0;
```

Flags Affected

The CF flag is set according to the result (see the "Operation" section above). The OF, SF, ZF, AF, and PF flags are set to 0.

C/C++ Compiler Intrinsic Equivalent

RDSEED int _rdseed16_step(unsigned short *);
RDSEED int _rdseed32_step(unsigned int *);
RDSEED int _rdseed64_step(unsigned __int64 *);

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.07H.00H:EBX.RDSEED[18] = 0.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.07H.00H:EBX.RDSEED[18] = 0.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.07H.00H:EBX.RDSEED[18] = 0.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.07H.00H:EBX.RDSEED[18] = 0.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.07H.00H:EBX.RDSEED[18] = 0.

RDSSPD/RDSSPQ—Read Shadow Stack Pointer

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 1E /1 (mod=11) RDSSPD r32	R	V/V	CET_SS	Copy low 32 bits of shadow stack pointer (SSP) to r32.
F3 REX.W 0F 1E /1 (mod=11) RDSSPQ r64	R	V/N.E.	CET_SS	Copies shadow stack pointer (SSP) to r64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
R	ModRM:r/m (w)	N/A	N/A	N/A

Description

Copies the current shadow stack pointer (SSP) register to the register destination. This opcode is a NOP when CET shadow stacks are not enabled and on processors that do not support CET.

Operation

```
IF CPL = 3
    IF CR4.CET & IA32_U_CET.SH_STK_EN
        IF (operand size is 64 bit)
            THEN
                Dest := SSP;
            ELSE
                Dest := SSP[31:0];
        FI;
    FI;
ELSE
    IF CR4.CET & IA32_S_CET.SH_STK_EN
        IF (operand size is 64 bit)
            THEN
                Dest := SSP;
            ELSE
                Dest := SSP[31:0];
        FI;
    FI;
FI;
```

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

RDSSPD__int32 _rdsspd_i32(void);
RDSSPQ__int64 _rdsspq_i64(void);

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

RDTSC—Read Time-Stamp Counter

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 31	RDTSC	ZO	Valid	Valid	Read time-stamp counter into EDX:EAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Reads the current value of the processor's time-stamp counter (a 64-bit MSR) into the EDX:EAX registers. The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.)

The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. See "Time-Stamp Counter" in Chapter 20 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B, for specific details of the time stamp counter behavior.

The time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSC instruction as follows. When the flag is clear, the RDTSC instruction can be executed at any privilege level; when the flag is set, the instruction can only be executed at privilege level 0.

The time-stamp counter can also be read with the RDMSR instruction, when executing at privilege level 0.

The RDTSC instruction is not a serializing instruction. It does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the read operation is performed. The following items may guide software seeking to order executions of RDTSC:

- If software requires RDTSC to be executed only after all previous instructions have executed and all previous loads are globally visible,[1] it can execute LFENCE immediately before RDTSC.
- If software requires RDTSC to be executed only after all previous instructions have executed and all previous loads and stores are globally visible, it can execute the sequence MFENCE;LFENCE immediately before RDTSC.
- If software requires RDTSC to be executed prior to execution of any subsequent instruction (including any memory accesses), it can execute the sequence LFENCE immediately after RDTSC.

This instruction was introduced by the Pentium processor.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 27 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C, for more information about the behavior of this instruction in VMX non-root operation.

Operation

```
IF (CR4.TSD = 0) or (CPL = 0) or (CR0.PE = 0)
    THEN EDX:EAX := TimeStampCounter;
    ELSE (* CR4.TSD = 1 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
        #GP(0);
FI;
```

Flags Affected

None.

1. A load is considered to become globally visible when the value to be loaded is determined.

Protected Mode Exceptions

#GP(0)	If the TSD flag in register CR4 is set and the CPL is greater than 0.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If the TSD flag in register CR4 is set.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

RDTSCP—Read Time-Stamp Counter and Processor ID

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 F9	RDTSCP	ZO	Valid	Valid	Read 64-bit time-stamp counter and IA32_TSC_AUX value into EDX:EAX and ECX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Reads the current value of the processor's time-stamp counter (a 64-bit MSR) into the EDX:EAX registers and also reads the value of the IA32_TSC_AUX MSR (address C0000103H) into the ECX register. The EDX register is loaded with the high-order 32 bits of the IA32_TSC MSR; the EAX register is loaded with the low-order 32 bits of the IA32_TSC MSR; and the ECX register is loaded with the low-order 32-bits of IA32_TSC_AUX MSR. On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX, RDX, and RCX are cleared.

The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. See "Time-Stamp Counter" in Chapter 20 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B, for specific details of the time stamp counter behavior.

The time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSCP instruction as follows. When the flag is clear, the RDTSCP instruction can be executed at any privilege level; when the flag is set, the instruction can only be executed at privilege level 0.

The RDTSCP instruction is not a serializing instruction, but it does wait until all previous instructions have executed and all previous loads are globally visible.[1] But it does not wait for previous stores to be globally visible, and subsequent instructions may begin execution before the read operation is performed. The following items may guide software seeking to order executions of RDTSCP:

- If software requires RDTSCP to be executed only after all previous stores are globally visible, it can execute MFENCE immediately before RDTSCP.
- If software requires RDTSCP to be executed prior to execution of any subsequent instruction (including any memory accesses), it can execute LFENCE immediately after RDTSCP.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 27 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C, for more information about the behavior of this instruction in VMX non-root operation.

Operation

```
IF (CR4.TSD = 0) or (CPL = 0) or (CR0.PE = 0)
    THEN
        EDX:EAX := TimeStampCounter;
        ECX := IA32_TSC_AUX[31:0];
    ELSE (* CR4.TSD = 1 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
        #GP(0);
FI;
```

Flags Affected

None.

1. A load is considered to become globally visible when the value to be loaded is determined.

Protected Mode Exceptions

#GP(0)	If the TSD flag in register CR4 is set and the CPL is greater than 0.
#UD	If the LOCK prefix is used.
	If CPUID.80000001H:EDX.RDTSCP[27] = 0.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.80000001H:EDX.RDTSCP[27] = 0.

Virtual-8086 Mode Exceptions

#GP(0)	If the TSD flag in register CR4 is set.
#UD	If the LOCK prefix is used.
	If CPUID.80000001H:EDX.RDTSCP[27] = 0.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

REP—Repeat String Operation (Prefix)

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description[1]
F3 6C	REP INS m8, DX	ZO	Valid	Valid	Input (count) bytes from port DX to (dest).
F3 6D	REP INS m16, DX	ZO	Valid	Valid	Input (count) words from port DX to (dest).
F3 6D	REP INS m32, DX	ZO	Valid	Valid	Input (count) doublewords from port DX to (dest).
F3 AC	REP LODS AL	ZO	Valid	Valid	Load (count) bytes from (src) to AL.
F3 AD	REP LODS AX	ZO	Valid	Valid	Load (count) words from (src) to AX.
F3 AD	REP LODS EAX	ZO	Valid	Valid	Load (count) doublewords from (src) to EAX.
F3 REX.W AD	REP LODS RAX	ZO	Valid	N.E.	Load (count) quadwords from (src) to RAX.
F3 A4	REP MOVS m8, m8	ZO	Valid	Valid	Move (count) bytes from (src) to (dest).
F3 A5	REP MOVS m16, m16	ZO	Valid	Valid	Move (count) words from (src) to (dest).
F3 A5	REP MOVS m32, m32	ZO	Valid	Valid	Move (count) doublewords from (src) to (dest).
F3 REX.W A5	REP MOVS m64, m64	ZO	Valid	N.E.	Move (count) quadwords from (src) to (dest).
F3 6E	REP OUTS DX, m8	ZO	Valid	Valid	Output (count) bytes from (src) to port DX.
F3 6F	REP OUTS DX, m16	ZO	Valid	Valid	Output (count) words from (src) to port DX.
F3 6F	REP OUTS DX, m32	ZO	Valid	Valid	Output (count) doublewords from (src) to port DX.
F3 AA	REP STOS m8	ZO	Valid	Valid	Fill (count) bytes at (dest) with AL.
F3 AB	REP STOS m16	ZO	Valid	Valid	Fill (count) words at (dest) with AX.
F3 AB	REP STOS m32	ZO	Valid	Valid	Fill (count) doublewords at (dest) with EAX.
F3 REX.W AB	REP STOS m64	ZO	Valid	N.E.	Fill (count) quadwords at (dest) with RAX.

NOTES:

1. See Description section for details on (count), (src), and (dest).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Repeats a string instruction the number of times specified in the count register. The count register is CX, ECX, or RCX, depending on the instruction's address size. The REP (repeat) mnemonic is a prefix that can be added to the INS, OUTS, MOVS, LODS, and STOS instructions.

The REP prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct. The REP prefixes causes the associated instruction to be repeated until the count in register is decremented to 0.

Each of the string instructions uses a source address, a destination address, or both. The source address is DS:SI, DS:ESI, or DS:RSI, depending on the instruction's address size; the DS segment may be overridden by an instruction prefix. The destination address is ES:DI, ES:EDI, or ES:RDI, depending on the instruction's address size; the ES segment may not be overridden. (Note that, in 64-bit mode, the base addresses of the CS, DS, ES, and SS segments are treated as zero.)

Similarly, the size of the count register is the instruction's address size. Thus, the default count register in 64-bit mode is RCX; REX.W has no effect on the address size and the count register. If 67H is used to override the default address size, the size of the count register is also overridden.

A repeating string operation can be suspended by an exception or interrupt. When this happens, the state of the registers is preserved to allow the string operation to be resumed upon a return from the exception or interrupt handler. The source and destination registers point to the next string elements to be operated on, the EIP register points to the string instruction, and the ECX register has the value it held following the last successful iteration of the instruction. This mechanism allows long string operations to proceed without affecting the interrupt response time of the system.

Use the REP INS and REP OUTS instructions with caution. Not all I/O ports can handle the rate at which these instructions execute. Note that a REP STOS instruction is the fastest way to initialize a large block of memory.

REP INS may read from the I/O port without writing to the memory location if an exception or VM exit occurs due to the write (e.g., #PF). If this would be problematic, for example because the I/O port read has side-effects, software should ensure the write to the memory location does not cause an exception or VM exit.

Operation

```
IF AddressSize = 16
   THEN
      Use CX for CountReg;
      Implicit Source/Dest operand for memory use of SI/DI;
   ELSE IF AddressSize = 64
      THEN Use RCX for CountReg;
      Implicit Source/Dest operand for memory use of RSI/RDI;
   ELSE
      Use ECX for CountReg;
      Implicit Source/Dest operand for memory use of ESI/EDI;
FI;
WHILE CountReg ≠ 0
    DO
         Service pending interrupts (if any);
         Execute associated string instruction;
         CountReg := (CountReg – 1);
    OD;
```

Flags Affected

None.

Exceptions (All Operating Modes)

Exceptions may be generated by an instruction associated with the prefix.

REPE/REPZ—Repeat String Operation While Zero (Prefix)

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description[1]
F3 A6	REPE CMPS m8, m8	ZO	Valid	Valid	Find nonmatching bytes in (src1) and (src2).
F3 A7	REPE CMPS m16, m16	ZO	Valid	Valid	Find nonmatching words in (src1) and (src2).
F3 A7	REPE CMPS m32, m32	ZO	Valid	Valid	Find nonmatching doublewords in (src1) and (src2).
F3 REX.W A7	REPE CMPS m64, m64	ZO	Valid	N.E.	Find nonmatching quadwords in (src1) and (src2).
F3 AE	REPE SCAS m8	ZO	Valid	Valid	Find non-AL byte starting at (src2).
F3 AF	REPE SCAS m16	ZO	Valid	Valid	Find non-AX word starting at (src2).
F3 AF	REPE SCAS m32	ZO	Valid	Valid	Find non-EAX doubleword starting at (src2).
F3 REX.W AF	REPE SCAS m64	ZO	Valid	N.E.	Find non-RAX quadword starting at (src2).

NOTES:

1. See Description section for details on (src1) and (src2).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Repeats a string instruction the number of times specified in the count register or until the ZF flag is clear. The count register is CX, ECX, or RCX, depending on the instruction's address size. The REPE (repeat while equal) and REPZ (repeat while zero) mnemonics are prefixes that can be added to the CMPS and SCAS instructions. (The REPZ prefix is a synonymous form of the REPE prefix.)

The REPE/REPZ prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct. These repeat prefixes cause the associated instruction to be repeated until the count in register is decremented to 0.

The REPE/REPZ prefixes check the state of the ZF flag after each iteration and terminate the repeat loop if the ZF flag is not set. When both termination conditions are tested, the cause of a repeat termination can be determined either by testing the count register with a JECXZ instruction or by testing the ZF flag (with a JZ, JNZ, or JNE instruction).

The ZF flag does not require initialization because both the CMPS and SCAS instructions affect the ZF flag according to the results of the comparisons they make.

Each of the string instructions uses one or two source addresses. The first source address is DS:SI, DS:ESI, or DS:RSI, depending on the instruction's address size; the DS segment may be overridden by an instruction prefix. The second source address is ES:DI, ES:EDI, or ES:RDI, depending on the instruction's address size; the ES segment may not be overridden. (Note that, in 64-bit mode, the base addresses of the CS, DS, ES, and SS segments are treated as zero.)

Similarly, the size of the count register is the instruction's address size. Thus, the default count register in 64-bit mode is RCX; REX.W has no effect on the address size and the count register. If 67H is used to override the default address size, the size of the count register is also overridden.

A repeating string operation can be suspended by an exception or interrupt. When this happens, the state of the registers is preserved to allow the string operation to be resumed upon a return from the exception or interrupt handler. The source and destination registers point to the next string elements to be operated on, the EIP register points to the string instruction, and the ECX register has the value it held following the last successful iteration of the instruction. This mechanism allows long string operations to proceed without affecting the interrupt response time of the system.

When a fault occurs during the execution of a CMPS or SCAS instruction that is prefixed with REPE or REPZ, the EFLAGS value is restored to the state prior to the execution of the instruction. Since the SCAS and CMPS instructions do not use EFLAGS as an input, the processor can resume the instruction after the page fault handler.

Operation

```
IF AddressSize = 16
   THEN
      Use CX for CountReg;
      Implicit Source/Dest operand for memory use of SI/DI;
   ELSE IF AddressSize = 64
      THEN Use RCX for CountReg;
      Implicit Source/Dest operand for memory use of RSI/RDI;
   ELSE
      Use ECX for CountReg;
      Implicit Source/Dest operand for memory use of ESI/EDI;
FI;
WHILE CountReg ≠ 0
    DO
        Service pending interrupts (if any);
        Execute associated string instruction;
        CountReg := (CountReg – 1);
        IF ZF = 0
            THEN exit WHILE loop;
        FI;
    OD;
```

Flags Affected

None by the prefixes; however, the CMPS and SCAS instructions do set the status flags in the EFLAGS register.

Exceptions (All Operating Modes)

Exceptions may be generated by an instruction associated with the prefix.

REPNE/REPNZ—Repeat String Operation While Not Zero (Prefix)

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description[1]
F2 A6	REPNE CMPS m8, m8	ZO	Valid	Valid	Find matching bytes in (src1) and (src2).
F2 A7	REPNE CMPS m16, m16	ZO	Valid	Valid	Find matching words in (src1) and (src2).
F2 A7	REPNE CMPS m32, m32	ZO	Valid	Valid	Find matching doublewords in (src1) and (src2).
F2 REX.W A7	REPNE CMPS m64, m64	ZO	Valid	N.E.	Find matching quadwords in (src1) and (src2).
F2 AE	REPNE SCAS m8	ZO	Valid	Valid	Find AL, starting at (src2).
F2 AF	REPNE SCAS m16	ZO	Valid	Valid	Find AX, starting at (src2).
F2 AF	REPNE SCAS m32	ZO	Valid	Valid	Find EAX, starting at (src2).
F2 REX.W AF	REPNE SCAS m64	ZO	Valid	N.E.	Find RAX, starting at (src2).

NOTES:

1. See Description section for details on (src1) and (src2).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Repeats a string instruction the number of times specified in the count register or until the ZF flag is set. The count register is CX, ECX, or RCX, depending on the instruction's address size. The REPNE (repeat while not equal) and REPNZ (repeat while not zero) mnemonics are prefixes that can be added to the CMPS and SCAS instructions. (The REPNZ prefix is a synonymous form of the REPNE prefix.)

The REPNE/REPNZ prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct. These repeat prefixes cause the associated instruction to be repeated until the count in register is decremented to 0.

The REPNE/REPNZ prefixes also check the state of the ZF flag after each iteration and terminate the repeat loop if the ZF flag is set. When both termination conditions are tested, the cause of a repeat termination can be determined either by testing the count register with a JECXZ instruction or by testing the ZF flag (with a JZ, JNZ, or JNE instruction).

The ZF flag does not require initialization because it is checked only after each execution of CMPS and SCAS, and those instructions update the ZF flag according to the results of the comparisons they make.

Each of the string instructions uses one or two source addresses. The first source address is DS:SI, DS:ESI, or DS:RSI, depending on the instruction's address size; the DS segment may be overridden by an instruction prefix. The second source address is ES:DI, ES:EDI, or ES:RDI, depending on the instruction's address size; the ES segment may not be overridden. (Note that, in 64-bit mode, the base addresses of the CS, DS, ES, and SS segments are treated as zero.)

Similarly, the size of the count register is the instruction's address size. Thus, the default count register in 64-bit mode is RCX; REX.W has no effect on the address size and the count register. If 67H is used to override the default address size, the size of the count register is also overridden.

A repeating string operation can be suspended by an exception or interrupt. When this happens, the state of the registers is preserved to allow the string operation to be resumed upon a return from the exception or interrupt handler. The source and destination registers point to the next string elements to be operated on, the EIP register points to the string instruction, and the ECX register has the value it held following the last successful iteration of the instruction. This mechanism allows long string operations to proceed without affecting the interrupt response time of the system.

When a fault occurs during the execution of a CMPS or SCAS instruction that is prefixed with REPNE or REPNZ, the EFLAGS value is restored to the state prior to the execution of the instruction. Since the SCAS and CMPS instructions do not use EFLAGS as an input, the processor can resume the instruction after the page fault handler.

Operation

```
IF AddressSize = 16
   THEN
      Use CX for CountReg;
      Implicit Source/Dest operand for memory use of SI/DI;
   ELSE IF AddressSize = 64
      THEN Use RCX for CountReg;
      Implicit Source/Dest operand for memory use of RSI/RDI;
   ELSE
      Use ECX for CountReg;
      Implicit Source/Dest operand for memory use of ESI/EDI;
FI;
WHILE CountReg ≠ 0
    DO
        Service pending interrupts (if any);
        Execute associated string instruction;
        CountReg := (CountReg – 1);
        IF CountReg = 0
            THEN exit WHILE loop; FI;
        IF ZF = 1
            THEN exit WHILE loop; FI;
    OD;
```

Flags Affected

None; however, the CMPS and SCAS instructions do set the status flags in the EFLAGS register.

Exceptions (All Operating Modes)

Exceptions may be generated by an instruction associated with the prefix.

RET—Return From Procedure

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
C3	RET	ZO	Valid	Valid	Near return to calling procedure.
CB	RET	ZO	Valid	Valid	Far return to calling procedure.
C2 iw	RET imm16	I	Valid	Valid	Near return to calling procedure and pop imm16 bytes from stack.
CA iw	RET imm16	I	Valid	Valid	Far return to calling procedure and pop imm16 bytes from stack.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A
I	imm16	N/A	N/A	N/A

Description

Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

The optional source operand specifies the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed. It must be used when the CALL instruction used to switch to a new procedure uses a call gate with a non-zero word count to access the new procedure. Here, the source operand for the RET instruction must specify the same number of bytes as is specified in the word count field of the call gate.

The RET instruction can be used to execute three different types of returns:

- **Near return** — A return to a calling procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment return.
- **Far return** — A return to a calling procedure located in a different segment than the current code segment, sometimes referred to as an intersegment return.
- **Inter-privilege-level far return** — A far return to a different privilege level than that of the currently executing program or procedure.

The inter-privilege-level return type can only be executed in protected mode. See the section titled "Calling Procedures Using Call and RET" in Chapter 6 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for detailed information on near, far, and inter-privilege-level returns.

When executing a near return, the processor pops the return instruction pointer (offset) from the top of the stack into the EIP register and begins program execution at the new instruction pointer. The CS register is unchanged.

When executing a far return, the processor pops the return instruction pointer from the top of the stack into the EIP register, then pops the segment selector from the top of the stack into the CS register. The processor then begins program execution in the new code segment at the new instruction pointer.

The mechanics of an inter-privilege-level far return are similar to an intersegment return, except that the processor examines the privilege levels and access rights of the code and stack segments being returned to determine if the control transfer is allowed to be made. The DS, ES, FS, and GS segment registers are cleared by the RET instruction during an inter-privilege-level return if they refer to segments that are not allowed to be accessed at the new privilege level. Since a stack switch also occurs on an inter-privilege level return, the ESP and SS registers are loaded from the stack.

If parameters are passed to the called procedure during an inter-privilege level call, the optional source operand must be used with the RET instruction to release the parameters on the return. Here, the parameters are released both from the called procedure's stack and the calling procedure's stack (that is, the stack being returned to).

In 64-bit mode, the default operation size of this instruction is the stack-address size, i.e., 64 bits. This applies to near returns, not far returns; the default operation size of far returns is 32 bits.

Refer to Chapter 6, "Procedure Calls, Interrupts, and Exceptions," and Chapter 18, "Control-flow Enforcement Technology (CET)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for CET details.

When FRED transitions are enabled, an execution of far RET that would change CPL causes a general-protection exception, as does an execution of far RET that would enter compatibility mode when CPL is 0.

Instruction ordering. Instructions following a far return may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the far return have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

Unlike near indirect CALL and near indirect JMP, the processor will not speculatively execute the next sequential instruction after a near RET unless that instruction is also the target of a jump or is a target in a branch predictor.

Operation

```
(* Near return *)
IF instruction = near return
    THEN;
        IF OperandSize = 32
            THEN
                IF top 4 bytes of stack not within stack limits
                    THEN #SS(0); FI;
                EIP := Pop();
                IF ShadowStackEnabled(CPL)
                    tempSsEIP = ShadowStackPop4B();
                    IF EIP != TempSsEIP
                        THEN #CP(NEAR_RET); FI;
                FI;
            ELSE
                IF OperandSize = 64
                    THEN
                        IF top 8 bytes of stack not within stack limits
                            THEN #SS(0); FI;
                        RIP := Pop();
                        IF ShadowStackEnabled(CPL)
                            tempSsEIP = ShadowStackPop8B();
                            IF RIP != tempSsEIP
                                THEN #CP(NEAR_RET); FI;
                        FI;
                    ELSE (* OperandSize = 16 *)
                        IF top 2 bytes of stack not within stack limits
                            THEN #SS(0); FI;
                        tempEIP := Pop();
                        tempEIP := tempEIP AND 0000FFFFH;
                        IF tempEIP not within code segment limits
                            THEN #GP(0); FI;
                        EIP := tempEIP;
                        IF ShadowStackEnabled(CPL)
                            tempSsEip = ShadowStackPop4B();
                            IF EIP != tempSsEIP
                                THEN #CP(NEAR_RET); FI;
                        FI;
                FI;
        FI;

        IF instruction has immediate operand
```

```
            THEN (* Release parameters from stack *)
                IF StackAddressSize = 32
                    THEN
                        ESP := ESP + SRC;
                    ELSE
                        IF StackAddressSize = 64
                            THEN
                                RSP := RSP + SRC;
                            ELSE (* StackAddressSize = 16 *)
                                SP := SP + SRC;
                        FI;
                FI;
        FI;
FI;

(* Real-address mode or virtual-8086 mode *)
IF ((PE = 0) or (PE = 1 AND VM = 1)) and instruction = far return
    THEN
        IF OperandSize = 32
            THEN
                IF top 8 bytes of stack not within stack limits
                    THEN #SS(0); FI;
                EIP := Pop();
                CS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            ELSE (* OperandSize = 16 *)
                IF top 4 bytes of stack not within stack limits
                    THEN #SS(0); FI;
                tempEIP := Pop();
                tempEIP := tempEIP AND 0000FFFFH;
                IF tempEIP not within code segment limits
                    THEN #GP(0); FI;
                EIP := tempEIP;
                CS := Pop(); (* 16-bit pop *)
        FI;
    IF instruction has immediate operand
        THEN (* Release parameters from stack *)
            SP := SP + (SRC AND FFFFH);
    FI;
FI;

(* Protected mode, not virtual-8086 mode *)
IF (PE = 1 and VM = 0 and IA32_EFER.LMA = 0) and instruction = far return
    THEN
        IF OperandSize = 32
            THEN
                IF second doubleword on stack is not within stack limits
                    THEN #SS(0); FI;
            ELSE (* OperandSize = 16 *)
                IF second word on stack is not within stack limits
                    THEN #SS(0); FI;
        FI;
        IF return code segment selector is NULL
            THEN #GP(0); FI;
        IF return code segment selector addresses descriptor beyond descriptor table limit
```

```
            THEN #GP(selector); FI;
        Obtain descriptor to which return code segment selector points from descriptor table;
        IF return code segment descriptor is not a code segment
            THEN #GP(selector); FI;
        IF return code segment selector RPL < CPL
            THEN #GP(selector); FI;
        IF return code segment descriptor is conforming and return code segment DPL > return code segment selector RPL
            THEN #GP(selector); FI;
        IF return code segment descriptor is non-conforming and return code segment DPL ≠ return code segment selector RPL
            THEN #GP(selector); FI;
        IF return code segment descriptor is not present
            THEN #NP(selector); FI:
        IF return code segment selector RPL > CPL
            THEN GOTO RETURN-TO-OUTER-PRIVILEGE-LEVEL;
            ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL;
        FI;
FI;

RETURN-TO-SAME-PRIVILEGE-LEVEL:
    IF the return instruction pointer is not within the return code segment limit
        THEN #GP(0); FI;
    IF OperandSize = 32
        THEN
            EIP := Pop();
            CS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
        ELSE (* OperandSize = 16 *)
            EIP := Pop();
            EIP := EIP AND 0000FFFFH;
            CS := Pop(); (* 16-bit pop *)
    FI;
    IF instruction has immediate operand
        THEN (* Release parameters from stack *)
            IF StackAddressSize = 32
                THEN
                    ESP := ESP + SRC;
                ELSE (* StackAddressSize = 16 *)
                    SP := SP + SRC;
            FI;
    FI;
    IF ShadowStackEnabled(CPL)
        (* SSP must be 8 byte aligned *)
        IF SSP AND 0x7 != 0
            THEN #CP(FAR-RET/IRET); FI;
        tempSsCS = shadow_stack_load 8 bytes from SSP+16;
        tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
        prevSSP = shadow_stack_load 8 bytes from SSP;
        SSP = SSP + 24;
        (* do a 64 bit-compare to check if any bits beyond bit 15 are set *)
        tempCS = CS; (* zero pad to 64 bit *)
        IF tempCS != tempSsCS
            THEN #CP(FAR-RET/IRET); FI;
        (* do a 64 bit-compare; pad CSBASE+RIP with 0 for 32 bit LIP*)
        IF CSBASE + RIP != tempSsLIP
            THEN #CP(FAR-RET/IRET); FI;
```

```
        (* prevSSP must be 4 byte aligned *)
        IF prevSSP AND 0x3 != 0
            THEN #CP(FAR-RET/IRET); FI;
        (* In legacy mode SSP must be in low 4GB *)
        IF prevSSP[63:32] != 0
            THEN #GP(0); FI;
        SSP := prevSSP
    FI;

RETURN-TO-OUTER-PRIVILEGE-LEVEL:
    IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize = 32)
    or top (8 + SRC) bytes of stack are not within stack limits (OperandSize = 16)
            THEN #SS(0); FI;
    Read return segment selector;
    IF stack segment selector is NULL
        THEN #GP(0); FI;
    IF return stack segment selector index is not within its descriptor table limits
        THEN #GP(selector); FI;
    Read segment descriptor pointed to by return segment selector;
    IF stack segment selector RPL ≠ RPL of the return code segment selector
    or stack segment is not a writable data segment
    or stack segment descriptor DPL ≠ RPL of the return code segment selector
            THEN #GP(selector); FI;
    IF stack segment not present
        THEN #SS(StackSegmentSelector); FI;
    IF the return instruction pointer is not within the return code segment limit
        THEN #GP(0); FI;
    IF OperandSize = 32
        THEN
            EIP := Pop();
            CS := Pop(); (* 32-bit pop, high-order 16 bits discarded; segment descriptor loaded *)
            CS(RPL) := ReturnCodeSegmentSelector(RPL);
            IF instruction has immediate operand
                THEN (* Release parameters from called procedure's stack *)
                    IF StackAddressSize = 32
                        THEN
                            ESP := ESP + SRC;
                        ELSE (* StackAddressSize = 16 *)
                            SP := SP + SRC;
                    FI;
            FI;
            tempESP := Pop();
            tempSS := Pop(); (* 32-bit pop, high-order 16 bits discarded; seg. descriptor loaded *)
        ELSE (* OperandSize = 16 *)
            EIP := Pop();
            EIP := EIP AND 0000FFFFH;
            CS := Pop(); (* 16-bit pop; segment descriptor loaded *)
            CS(RPL) := ReturnCodeSegmentSelector(RPL);
            IF instruction has immediate operand
                THEN (* Release parameters from called procedure's stack *)
                    IF StackAddressSize = 32
                        THEN
                            ESP := ESP + SRC;
                        ELSE (* StackAddressSize = 16 *)
```

```
                        SP := SP + SRC;
                    FI;
            FI;
            tempESP := Pop();
            tempSS := Pop(); (* 16-bit pop; segment descriptor loaded *)
        FI;
    IF ShadowStackEnabled(CPL)
        (* check if 8 byte aligned *)
        IF SSP AND 0x7 != 0
            THEN #CP(FAR-RET/IRET); FI;
        IF ReturnCodeSegmentSelector(RPL) !=3
            THEN
                tempSsCS = shadow_stack_load 8 bytes from SSP+16;
                tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
                tempSSP = shadow_stack_load 8 bytes from SSP;
                SSP = SSP + 24;
                (* Do 64 bit compare to detect bits beyond 15 being set *)
                tempCS = CS; (* zero extended to 64 bit *)
                IF tempCS != tempSsCS
                    THEN #CP(FAR-RET/IRET); FI;
                (* Do 64 bit compare; pad CSBASE+RIP with 0 for 32 bit LA *)
                IF CSBASE + RIP != tempSsLIP
                    THEN #CP(FAR-RET/IRET); FI;
                (* check if 4 byte aligned *)
                IF tempSSP AND 0x3 != 0
                    THEN #CP(FAR-RET/IRET); FI;
        FI;
    FI;
        tempOldCPL = CPL;

        CPL := ReturnCodeSegmentSelector(RPL);
        ESP := tempESP;
        SS := tempSS;
        tempOldSSP = SSP;
        IF ShadowStackEnabled(CPL)
            IF CPL = 3
                THEN tempSSP := IA32_PL3_SSP; FI;
            IF tempSSP[63:32] != 0
                THEN #GP(0); FI;
            SSP := tempSSP
        FI;
        (* Now past all faulting points; safe to free the token. The token free is done using the old SSP
         * and using a supervisor override as old CPL was a supervisor privilege level *)
        IF ShadowStackEnabled(tempOldCPL)
            expected_token_value = tempOldSSP | BUSY_BIT (* busy bit - bit position 0 - must be set *)
            new_token_value = tempOldSSP                (* clear the busy bit *)
            shadow_stack_lock_cmpxchg8b(tempOldSSP, new_token_value, expected_token_value)
        FI;
    FI;

    FOR each SegReg in (ES, FS, GS, and DS)
        DO
            tempDesc := descriptor cache for SegReg (* hidden part of segment register *)
            IF (SegmentSelector == NULL) OR (tempDesc(DPL) < CPL AND tempDesc(Type) is (data or non-conforming code)))
```

```
                THEN (* Segment register invalid *)
                    SegmentSelector := 0; (*Segment selector becomes null*)
            FI;
        OD;

    IF instruction has immediate operand
        THEN (* Release parameters from calling procedure's stack *)
            IF StackAddressSize = 32
                THEN
                    ESP := ESP + SRC;
                ELSE (* StackAddressSize = 16 *)
                    SP := SP + SRC;
            FI;
    FI;

(* IA-32e Mode *)
    IF (PE = 1 and VM = 0 and IA32_EFER.LMA = 1) and instruction = far return
        THEN
            IF OperandSize = 32
                THEN
                    IF second doubleword on stack is not within stack limits
                        THEN #SS(0); FI;
                    IF first or second doubleword on stack is not in canonical space
                        THEN #SS(0); FI;
                ELSE
                    IF OperandSize = 16
                        THEN
                            IF second word on stack is not within stack limits
                                THEN #SS(0); FI;
                            IF first or second word on stack is not in canonical space
                                THEN #SS(0); FI;
                        ELSE (* OperandSize = 64 *)
                            IF first or second quadword on stack is not in canonical space
                                THEN #SS(0); FI;
                    FI
            FI;
        IF return code segment selector is NULL
            THEN GP(0); FI;
        IF return code segment selector addresses descriptor beyond descriptor table limit
            THEN GP(selector); FI;
        IF return code segment selector addresses descriptor in non-canonical space
            THEN GP(selector); FI;
        Obtain descriptor to which return code segment selector points from descriptor table;
        IF return code segment descriptor is not a code segment
            THEN #GP(selector); FI;
        IF return code segment descriptor has L-bit = 1 and D-bit = 1
            THEN #GP(selector); FI;
        IF return code segment selector RPL < CPL or (CR4.FRED = 1 and return code segment selector RPL > CPL)
            THEN #GP(selector); FI;
        IF return code segment descriptor is conforming and return code segment DPL > return code segment selector RPL
            THEN #GP(selector); FI;
        IF return code segment descriptor is non-conforming and return code segment DPL ≠ return code segment selector RPL
            THEN #GP(selector); FI;
        IF CR4.FRED = 1 and CPL = 0 and L-bit is 0 in return code segment descriptor
```

```
            THEN #GP(selector); FI;
        IF return code segment descriptor is not present
            THEN #NP(selector); FI:
        IF return code segment selector RPL > CPL
            THEN GOTO IA-32E-MODE-RETURN-TO-OUTER-PRIVILEGE-LEVEL;
            ELSE GOTO IA-32E-MODE-RETURN-TO-SAME-PRIVILEGE-LEVEL;
        FI;
    FI;

IA-32E-MODE-RETURN-TO-SAME-PRIVILEGE-LEVEL:
IF the return instruction pointer is not within the return code segment limit
    THEN #GP(0); FI;
IF the return instruction pointer is not within canonical address space
    THEN #GP(0); FI;
IF OperandSize = 32
    THEN
        EIP := Pop();
        CS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
    ELSE
        IF OperandSize = 16
            THEN
                EIP := Pop();
                EIP := EIP AND 0000FFFFH;
                CS := Pop(); (* 16-bit pop *)
            ELSE (* OperandSize = 64 *)
                RIP := Pop();
                CS := Pop(); (* 64-bit pop, high-order 48 bits discarded *)
        FI;
FI;
IF instruction has immediate operand
    THEN (* Release parameters from stack *)
        IF StackAddressSize = 32
            THEN
                ESP := ESP + SRC;
            ELSE
                IF StackAddressSize = 16
                    THEN
                        SP := SP + SRC;
                    ELSE (* StackAddressSize = 64 *)
                        RSP := RSP + SRC;
                FI;
        FI;
FI;
IF ShadowStackEnabled(CPL)
    IF SSP AND 0x7 != 0 (* check if aligned to 8 bytes *)
        THEN #CP(FAR-RET/IRET); FI;
    tempSsCS = shadow_stack_load 8 bytes from SSP+16;
    tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
    tempSSP = shadow_stack_load 8 bytes from SSP;
    SSP = SSP + 24;
    tempCS = CS; (* zero padded to 64 bit *)
    IF tempCS != tempSsCS (* 64 bit compare; CS zero padded to 64 bits *)
        THEN #CP(FAR-RET/IRET); FI;
    IF CSBASE + RIP != tempSsLIP (* 64 bit compare *)
```

```
            THEN #CP(FAR-RET/IRET); FI;
        IF tempSSP AND 0x3 != 0 (* check if aligned to 4 bytes *)
            THEN #CP(FAR-RET/IRET); FI;
        IF (CS.L = 0 AND tempSSP[63:32] != 0) OR
           (CS.L = 1 AND tempSSP is not canonical relative to the current paging mode)
            THEN #GP(0); FI;
        SSP := tempSSP
FI;

IA-32E-MODE-RETURN-TO-OUTER-PRIVILEGE-LEVEL:
IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize = 32)
or top (8 + SRC) bytes of stack are not within stack limits (OperandSize = 16)
    THEN #SS(0); FI;
IF top (16 + SRC) bytes of stack are not in canonical address space (OperandSize =32)
or top (8 + SRC) bytes of stack are not in canonical address space (OperandSize = 16)
or top (32 + SRC) bytes of stack are not in canonical address space (OperandSize = 64)
    THEN #SS(0); FI;
Read return stack segment selector;
IF stack segment selector is NULL
    THEN
        IF new CS descriptor L-bit = 0
            THEN #GP(selector);
        IF stack segment selector RPL = 3
            THEN #GP(selector);
FI;
IF return stack segment descriptor is not within descriptor table limits
        THEN #GP(selector); FI;
IF return stack segment descriptor is in non-canonical address space
        THEN #GP(selector); FI;
Read segment descriptor pointed to by return segment selector;
IF stack segment selector RPL ≠ RPL of the return code segment selector
or stack segment is not a writable data segment
or stack segment descriptor DPL ≠ RPL of the return code segment selector
    THEN #GP(selector); FI;
IF stack segment not present
    THEN #SS(StackSegmentSelector); FI;
IF the return instruction pointer is not within the return code segment limit
    THEN #GP(0); FI:
IF the return instruction pointer is not within canonical address space
    THEN #GP(0); FI;
IF OperandSize = 32
    THEN
        EIP := Pop();
        CS := Pop(); (* 32-bit pop, high-order 16 bits discarded, segment descriptor loaded *)
        CS(RPL) := ReturnCodeSegmentSelector(RPL);
        IF instruction has immediate operand
            THEN (* Release parameters from called procedure's stack *)
                IF StackAddressSize = 32
                    THEN
                        ESP := ESP + SRC;
                    ELSE
                        IF StackAddressSize = 16
                            THEN
                                SP := SP + SRC;
```

```
                                ELSE (* StackAddressSize = 64 *)
                                    RSP := RSP + SRC;
                                FI;
                        FI;
                FI;
                tempESP := Pop();
                tempSS := Pop(); (* 32-bit pop, high-order 16 bits discarded, segment descriptor loaded *)
        ELSE
                IF OperandSize = 16
                        THEN
                                EIP := Pop();
                                EIP := EIP AND 0000FFFFH;
                                CS := Pop(); (* 16-bit pop; segment descriptor loaded *)
                                CS(RPL) := ReturnCodeSegmentSelector(RPL);
                                IF instruction has immediate operand
                                        THEN (* Release parameters from called procedure's stack *)
                                                IF StackAddressSize = 32
                                                        THEN
                                                                ESP := ESP + SRC;
                                                        ELSE
                                                                IF StackAddressSize = 16
                                                                        THEN
                                                                                SP := SP + SRC;
                                                                        ELSE (* StackAddressSize = 64 *)
                                                                                RSP := RSP + SRC;
                                                                FI;
                                                FI;
                                FI;
                                tempESP := Pop();
                                tempSS := Pop(); (* 16-bit pop; segment descriptor loaded *)
                        ELSE (* OperandSize = 64 *)
                                RIP := Pop();
                                CS := Pop(); (* 64-bit pop; high-order 48 bits discarded; seg. descriptor loaded *)
                                CS(RPL) := ReturnCodeSegmentSelector(RPL);
                                IF instruction has immediate operand
                                        THEN (* Release parameters from called procedure's stack *)
                                                RSP := RSP + SRC;
                                FI;
                                tempESP := Pop();
                                tempSS := Pop(); (* 64-bit pop; high-order 48 bits discarded; seg. desc. loaded *)
                FI;
FI;

IF ShadowStackEnabled(CPL)
        (* check if 8 byte aligned *)
        IF SSP AND 0x7 != 0
                THEN #CP(FAR-RET/IRET); FI;
        IF ReturnCodeSegmentSelector(RPL) !=3
                THEN
                        tempSsCS = shadow_stack_load 8 bytes from SSP+16;
                        tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
                        tempSSP = shadow_stack_load 8 bytes from SSP;
                        SSP = SSP + 24;
                        (* Do 64 bit compare to detect bits beyond 15 being set *)
```

```
                tempCS = CS; (* zero padded to 64 bit *)
                IF tempCS != tempSsCS
                    THEN #CP(FAR-RET/IRET); FI;
                (* Do 64 bit compare; pad CSBASE+RIP with 0 for 32 bit LIP *)
                IF CSBASE + RIP != tempSsLIP
                    THEN #CP(FAR-RET/IRET); FI;
                (* check if 4 byte aligned *)
                IF tempSSP AND 0x3 != 0
                    THEN #CP(FAR-RET/IRET); FI;
    FI;
FI;
tempOldCPL = CPL;
CPL := ReturnCodeSegmentSelector(RPL);
ESP := tempESP;
SS := tempSS;
tempOldSSP = SSP;
IF ShadowStackEnabled(CPL)
    IF CPL = 3
        THEN tempSSP := IA32_PL3_SSP; FI;
    IF (CS.L = 0 AND tempSSP[63:32] != 0) OR
       (CS.L = 1 AND tempSSP is not canonical relative to the current paging mode)
        THEN #GP(0); FI;
    SSP := tempSSP
FI;
(* Now past all faulting points; safe to free the token. The token free is done using the old SSP
* and using a supervisor override as old CPL was a supervisor privilege level *)
IF ShadowStackEnabled(tempOldCPL)
    expected_token_value = tempOldSSP | BUSY_BIT        (* busy bit - bit position 0 - must be set *)
    new_token_value = tempOldSSP                        (* clear the busy bit *)
    shadow_stack_lock_cmpxchg8b(tempOldSSP, new_token_value, expected_token_value)
FI;

FOR each of segment register (ES, FS, GS, and DS)
    DO
        IF segment register points to data or non-conforming code segment
        and CPL > segment descriptor DPL; (* DPL in hidden part of segment register *)
            THEN SegmentSelector := 0; (* SegmentSelector invalid *)
        FI;
    OD;

IF instruction has immediate operand
    THEN (* Release parameters from calling procedure's stack *)
        IF StackAddressSize = 32
            THEN
                ESP := ESP + SRC;
            ELSE
                IF StackAddressSize = 16
                    THEN
                        SP := SP + SRC;
                    ELSE (* StackAddressSize = 64 *)
                        RSP := RSP + SRC;
                FI;
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the return code or stack segment selector is NULL.
	If the return instruction pointer is not within the return code segment limit.
	If returning to 32-bit or compatibility mode and the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is beyond 4GB.
#GP(selector)	If the RPL of the return code segment selector is less then the CPL.
	If the return code or stack segment selector index is not within its descriptor table limits.
	If the return code segment descriptor does not indicate a code segment.
	If the return code segment is non-conforming and the segment selector's DPL is not equal to the RPL of the code segment's segment selector
	If the return code segment is conforming and the segment selector's DPL greater than the RPL of the code segment's segment selector
	If the stack segment is not a writable data segment.
	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
#SS(0)	If the top bytes of stack are not within stack limits.
	If the return stack segment is not present.
#NP(selector)	If the return code segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory access occurs when the CPL is 3 and alignment checking is enabled.
#CP(Far-RET/IRET)	If the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is not 4 byte aligned.
	If return instruction pointer from stack and shadow stack do not match.

Real-Address Mode Exceptions

#GP	If the return instruction pointer is not within the return code segment limit
#SS	If the top bytes of stack are not within stack limits.

Virtual-8086 Mode Exceptions

#GP(0)	If the return instruction pointer is not within the return code segment limit
#SS(0)	If the top bytes of stack are not within stack limits.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory access occurs when alignment checking is enabled.

Compatibility Mode Exceptions

Same as 64-bit mode exceptions.

64-Bit Mode Exceptions

#GP(0)	If the return instruction pointer is non-canonical.
	If the return instruction pointer is not within the return code segment limit.
	If the stack segment selector is NULL going back to compatibility mode.
	If the stack segment selector is NULL going back to CPL3 64-bit mode.
	If a NULL stack segment selector RPL is not equal to CPL going back to non-CPL3 64-bit mode.
	If the return code segment selector is NULL.
	If returning to 32-bit or compatibility mode and the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is beyond 4GB.
#GP(selector)	If the proposed segment descriptor for a code segment does not indicate it is a code segment.
	If the proposed new code segment descriptor has both the D-bit and L-bit set.
	If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.
	If CPL is greater than the RPL of the code segment selector.
	If FRED transitions are enabled and CPL is less than the RPL of the code segment selector.
	If FRED transitions are enabled and CPL = 0 and far RET would enter compatibility mode.
	If the DPL of a conforming-code segment is greater than the return code segment selector RPL.
	If a segment selector index is outside its descriptor table limits.
	If a segment descriptor memory address is non-canonical.
	If the stack segment is not a writable data segment.
	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
#SS(0)	If an attempt to pop a value off the stack violates the SS limit.
	If an attempt to pop a value off the stack causes a non-canonical address to be referenced.
#NP(selector)	If the return code or stack segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#CP(Far-RET/IRET)	If the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is not 4 byte aligned.
	If return instruction pointer from stack and shadow stack do not match.

RORX — Rotate Right Logical Without Affecting Flags

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.LZ.F2.0F3A.W0 F0 /r ib RORX r32, r/m32, imm8	RMI	V/V	BMI2	Rotate 32-bit r/m32 right imm8 times and load into r32 without affecting arithmetic flags.
VEX.LZ.F2.0F3A.W1 F0 /r ib RORX r64, r/m64, imm8	RMI	V/N.E.	BMI2	Rotate 64-bit r/m64 right imm8 times and load into r64 without affecting arithmetic flags.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Rotates the bits of source operand right by the count value specified in imm8 without affecting arithmetic flags. The result is written to the 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

```
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));
FI;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-29, “Type 13 Class Exception Conditions.”

ROUNDPD—Round Packed Double Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 09 /r ib ROUNDPD xmm1, xmm2/m128, imm8	RMI	V/V	SSE4_1	Round packed double precision floating-point values in xmm2/m128 and place the result in xmm1. The rounding mode is determined by imm8.
VEX.128.66.0F3A.WIG 09 /r ib VROUNDPD xmm1, xmm2/m128, imm8	RMI	V/V	AVX	Round packed double precision floating-point values in xmm2/m128 and place the result in xmm1. The rounding mode is determined by imm8.
VEX.256.66.0F3A.WIG 09 /r ib VROUNDPD ymm1, ymm2/m256, imm8	RMI	V/V	AVX	Round packed double precision floating-point values in ymm2/m256 and place the result in ymm1. The rounding mode is determined by imm8.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Round the 2 double precision floating-point values in the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the results in the destination operand (first operand). The rounding process rounds each input floating-point value to an integer value and returns the integer result as a double precision floating-point value.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-24. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-21 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to `1 then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

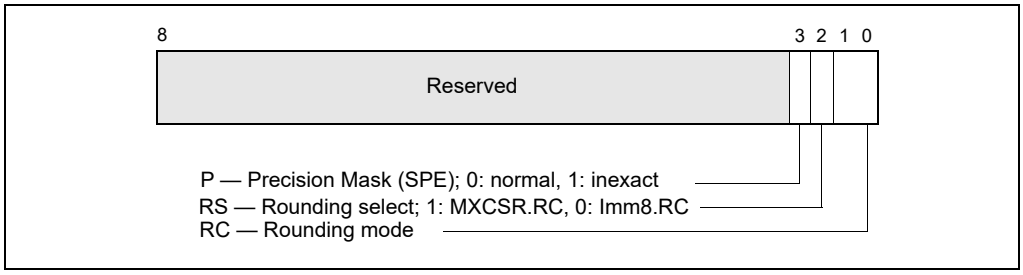


Figure 4-24. Bit Control Fields of Immediate Byte for ROUNDxx Instruction

Table 4-21. Rounding Modes and Encoding of Rounding Control (RC) Field

Rounding Mode	RC Field Setting	Description
Round to nearest (even)	00B	Rounded result is the closest to the infinitely precise result. If two values are equally close, the result is the even value (i.e., the integer value with the least-significant bit of zero).
Round down (toward −∞)	01B	Rounded result is closest to but no greater than the infinitely precise result.
Round up (toward +∞)	10B	Rounded result is closest to but no less than the infinitely precise result.
Round toward zero (Truncate)	11B	Rounded result is closest to but no greater in absolute value than the infinitely precise result.

Operation

```
IF (imm[2] = '1)
    THEN    // rounding mode is determined by MXCSR.RC
        DEST[63:0] := ConvertDPFPToInteger_M(SRC[63:0]);
        DEST[127:64] := ConvertDPFPToInteger_M(SRC[127:64]);
    ELSE    // rounding mode is determined by IMM8.RC
        DEST[63:0] := ConvertDPFPToInteger_Imm(SRC[63:0]);
        DEST[127:64] := ConvertDPFPToInteger_Imm(SRC[127:64]);
FI
```

ROUNDPD (128-bit Legacy SSE Version)

```
DEST[63:0] := RoundToInteger(SRC[63:0]], ROUND_CONTROL)
DEST[127:64] := RoundToInteger(SRC[127:64]], ROUND_CONTROL)
DEST[MAXVL-1:128] (Unmodified)
```

VROUNDPD (VEX.128 Encoded Version)

```
DEST[63:0] := RoundToInteger(SRC[63:0]], ROUND_CONTROL)
DEST[127:64] := RoundToInteger(SRC[127:64]], ROUND_CONTROL)
DEST[MAXVL-1:128] := 0
```

VROUNDPD (VEX.256 Encoded Version)

```
DEST[63:0] := RoundToInteger(SRC[63:0], ROUND_CONTROL)
DEST[127:64] := RoundToInteger(SRC[127:64]], ROUND_CONTROL)
DEST[191:128] := RoundToInteger(SRC[191:128]], ROUND_CONTROL)
DEST[255:192] := RoundToInteger(SRC[255:192] ], ROUND_CONTROL)
```

Intel C/C++ Compiler Intrinsic Equivalent

__m128 _mm_round_pd(__m128d s1, int iRoundMode);
__m128 _mm_floor_pd(__m128d s1);
__m128 _mm_ceil_pd(__m128d s1)
__m256 _mm256_round_pd(__m256d s1, int iRoundMode);
__m256 _mm256_floor_pd(__m256d s1);
__m256 _mm256_ceil_pd(__m256d s1)

SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN).

Precision (signaled only if imm[3] = `0; if imm[3] = `1, then the Precision Mask in the MXCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDPD.

Other Exceptions

See Table 2-19, "Type 2 Class Exception Conditions," additionally:

#UD	If VEX.vvvv ≠ 1111B.

ROUNDPS—Round Packed Single Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 08 /r ib ROUNDPS xmm1, xmm2/m128, imm8	RMI	V/V	SSE4_1	Round packed single precision floating-point values in xmm2/m128 and place the result in xmm1. The rounding mode is determined by imm8.
VEX.128.66.0F3A.WIG 08 /r ib VROUNDPS xmm1, xmm2/m128, imm8	RMI	V/V	AVX	Round packed single precision floating-point values in xmm2/m128 and place the result in xmm1. The rounding mode is determined by imm8.
VEX.256.66.0F3A.WIG 08 /r ib VROUNDPS ymm1, ymm2/m256, imm8	RMI	V/V	AVX	Round packed single precision floating-point values in ymm2/m256 and place the result in ymm1. The rounding mode is determined by imm8.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Round the 4 single precision floating-point values in the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the results in the destination operand (first operand). The rounding process rounds each input floating-point value to an integer value and returns the integer result as a single precision floating-point value.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-24. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-21 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to `1 then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

```
IF (imm[2] = '1)
    THEN    // rounding mode is determined by MXCSR.RC
        DEST[31:0] := ConvertSPFPToInteger_M(SRC[31:0]);
        DEST[63:32] := ConvertSPFPToInteger_M(SRC[63:32]);
        DEST[95:64] := ConvertSPFPToInteger_M(SRC[95:64]);
        DEST[127:96] := ConvertSPFPToInteger_M(SRC[127:96]);
    ELSE    // rounding mode is determined by IMM8.RC
```

```
        DEST[31:0] := ConvertSPFPToInteger_Imm(SRC[31:0]);
        DEST[63:32] := ConvertSPFPToInteger_Imm(SRC[63:32]);
        DEST[95:64] := ConvertSPFPToInteger_Imm(SRC[95:64]);
        DEST[127:96] := ConvertSPFPToInteger_Imm(SRC[127:96]);
FI;
```

ROUNDPS(128-bit Legacy SSE Version)

```
DEST[31:0] := RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] := RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] := RoundToInteger(SRC[95:64]], ROUND_CONTROL)
DEST[127:96] := RoundToInteger(SRC[127:96]], ROUND_CONTROL)
DEST[MAXVL-1:128] (Unmodified)
```

VROUNDPS (VEX.128 Encoded Version)

```
DEST[31:0] := RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] := RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] := RoundToInteger(SRC[95:64]], ROUND_CONTROL)
DEST[127:96] := RoundToInteger(SRC[127:96]], ROUND_CONTROL)
DEST[MAXVL-1:128] := 0
```

VROUNDPS (VEX.256 Encoded Version)

```
DEST[31:0] := RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] := RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] := RoundToInteger(SRC[95:64]], ROUND_CONTROL)
DEST[127:96] := RoundToInteger(SRC[127:96]], ROUND_CONTROL)
DEST[159:128] := RoundToInteger(SRC[159:128]], ROUND_CONTROL)
DEST[191:160] := RoundToInteger(SRC[191:160]], ROUND_CONTROL)
DEST[223:192] := RoundToInteger(SRC[223:192] ], ROUND_CONTROL)
DEST[255:224] := RoundToInteger(SRC[255:224] ], ROUND_CONTROL)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
__m128 _mm_round_ps(__m128 s1, int iRoundMode);
__m128 _mm_floor_ps(__m128 s1);
__m128 _mm_ceil_ps(__m128 s1)
__m256 _mm256_round_ps(__m256 s1, int iRoundMode);
__m256 _mm256_floor_ps(__m256 s1);
__m256 _mm256_ceil_ps(__m256 s1)
```

SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN).

Precision (signaled only if imm[3] = ‘0; if imm[3] = ‘1, then the Precision Mask in the MXCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDPS.

Other Exceptions

See Table 2-19, “Type 2 Class Exception Conditions,” additionally:

#UD	If VEX.vvvv ≠ 1111B.

ROUNDSD—Round Scalar Double Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 0B /r ib ROUNDSD xmm1, xmm2/m64, imm8	RMI	V/V	SSE4_1	Round the low packed double precision floating-point value in xmm2/m64 and place the result in xmm1. The rounding mode is determined by imm8.
VEX.LIG.66.0F3A.WIG 0B /r ib VROUNDSD xmm1, xmm2, xmm3/m64, imm8	RVMI	V/V	AVX	Round the low packed double precision floating-point value in xmm3/m64 and place the result in xmm1. The rounding mode is determined by imm8. Upper packed double precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Round the double precision floating-point value in the lower qword of the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the result in the destination operand (first operand). The rounding process rounds a double precision floating-point input to an integer value and returns the integer result as a double precision floating-point value in the lowest position. The upper double precision floating-point value in the destination is retained.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-24. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-21 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to `1 then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

Operation

```
IF (imm[2] = '1)
    THEN    // rounding mode is determined by MXCSR.RC
        DEST[63:0] := ConvertDPFPToInteger_M(SRC[63:0]);
    ELSE    // rounding mode is determined by IMM8.RC
        DEST[63:0] := ConvertDPFPToInteger_Imm(SRC[63:0]);
FI;
DEST[127:63] remains unchanged ;
```

ROUNDSD (128-bit Legacy SSE Version)

```
DEST[63:0] := RoundToInteger(SRC[63:0], ROUND_CONTROL)
DEST[MAXVL-1:64] (Unmodified)
```

VROUNDSD (VEX.128 Encoded Version)

```
DEST[63:0] := RoundToInteger(SRC2[63:0], ROUND_CONTROL)
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

ROUNDSD __m128d mm_round_sd(__m128d dst, __m128d s1, int iRoundMode);
ROUNDSD __m128d mm_floor_sd(__m128d dst, __m128d s1);
ROUNDSD __m128d mm_ceil_sd(__m128d dst, __m128d s1);

SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN).

Precision (signaled only if imm[3] = '0; if imm[3] = '1, then the Precision Mask in the MXCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDSD.

Other Exceptions

See Table 2-20, "Type 3 Class Exception Conditions."

ROUNDSS—Round Scalar Single Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 0A /r ib ROUNDSS xmm1, xmm2/m32, imm8	RMI	V/V	SSE4_1	Round the low packed single precision floating-point value in xmm2/m32 and place the result in xmm1. The rounding mode is determined by imm8.
VEX.LIG.66.0F3A.WIG 0A /r ib VROUNDSS xmm1, xmm2, xmm3/m32, imm8	RVMI	V/V	AVX	Round the low packed single precision floating-point value in xmm3/m32 and place the result in xmm1. The rounding mode is determined by imm8. Also, upper packed single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Round the single precision floating-point value in the lowest dword of the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the result in the destination operand (first operand). The rounding process rounds a single precision floating-point input to an integer value and returns the result as a single precision floating-point value in the lowest position. The upper three single precision floating-point values in the destination are retained.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-24. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-21 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to `1 then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

Operation

```
IF (imm[2] = '1)
    THEN    // rounding mode is determined by MXCSR.RC
        DEST[31:0] := ConvertSPFPToInteger_M(SRC[31:0]);
    ELSE    // rounding mode is determined by IMM8.RC
        DEST[31:0] := ConvertSPFPToInteger_Imm(SRC[31:0]);
FI;
DEST[127:32] remains unchanged ;
```

ROUNDSS (128-bit Legacy SSE Version)

```
DEST[31:0] := RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[MAXVL-1:32] (Unmodified)
```

VROUNDSS (VEX.128 Encoded Version)

```
DEST[31:0] := RoundToInteger(SRC2[31:0], ROUND_CONTROL)
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

ROUNDSS __m128 mm_round_ss(__m128 dst, __m128 s1, int iRoundMode);
ROUNDSS __m128 mm_floor_ss(__m128 dst, __m128 s1);
ROUNDSS __m128 mm_ceil_ss(__m128 dst, __m128 s1);

SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN).

Precision (signaled only if imm[3] = '0; if imm[3] = '1, then the Precision Mask in the MXCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDSS.

Other Exceptions

See Table 2-20, "Type 3 Class Exception Conditions."

RSM—Resume From System Management Mode

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F AA	RSM	ZO	Valid	Valid	Resume operation of interrupted program.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Returns program control from system management mode (SMM) to the application program or operating-system procedure that was interrupted when the processor received an SMM interrupt. The processor's state is restored from the dump created upon entering SMM. If the processor detects invalid state information during state restoration, it enters the shutdown state. The following invalid information can cause a shutdown:

- Any reserved bit of CR4 is set to 1.
- Any illegal combination of bits in CR0, such as (PG=1 and PE=0) or (NW=1 and CD=0).
- (Intel Pentium and Intel486™ processors only.) The value stored in the state dump base field is not a 32-KByte aligned address.

The contents of the model-specific registers are not affected by a return from SMM.

The SMM state map used by RSM supports resuming processor context for non-64-bit modes and 64-bit mode.

See Chapter 34, "System Management Mode," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C, for more information about SMM and the behavior of the RSM instruction.

Operation

```
ReturnFromSMM;
IF (IA-32e mode supported) or (CPUID DisplayFamily_DisplayModel = 06H_0CH )
    THEN
        ProcessorState := Restore(SMMDump(IA-32e SMM STATE MAP));
    Else
        ProcessorState := Restore(SMMDump(Non-32-Bit-Mode SMM STATE MAP));
FI
```

Flags Affected

All.

Protected Mode Exceptions

#UD	If an attempt is made to execute this instruction when the processor is not in SMM.
	If the LOCK prefix is used.
#GP(0)	If CPL > 0 and IA32_CORE_CAPABILITIES.RSM_IN_CPL0_ONLY[bit 3] = 1.

Real-Address Mode Exceptions

#UD	If an attempt is made to execute this instruction when the processor is not in SMM.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD	If an attempt is made to execute this instruction when the processor is not in SMM.
	If the LOCK prefix is used.

#GP(0)	If IA32_CORE_CAPABILITIES.RSM_IN_CPL0_ONLY[bit 3] = 1.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

RSQRTPS—Compute Reciprocals of Square Roots of Packed Single Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 52 /r RSQRTPS xmm1, xmm2/m128	RM	V/V	SSE	Computes the approximate reciprocals of the square roots of the packed single precision floating-point values in xmm2/m128 and stores the results in xmm1.
VEX.128.0F.WIG 52 /r VRSQRTPS xmm1, xmm2/m128	RM	V/V	AVX	Computes the approximate reciprocals of the square roots of packed single precision values in xmm2/mem and stores the results in xmm1.
VEX.256.0F.WIG 52 /r VRSQRTPS ymm1, ymm2/m256	RM	V/V	AVX	Computes the approximate reciprocals of the square roots of packed single precision values in ymm2/mem and stores the results in ymm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Performs a SIMD computation of the approximate reciprocals of the square roots of the four packed single precision floating-point values in the source operand (second operand) and stores the packed single precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD single precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RSQRTPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). When a source value is a negative value (other than –0.0), a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

RSQRTPS (128-bit Legacy SSE Version)

```
DEST[31:0] := APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] := APPROXIMATE(1/SQRT(SRC1[63:32]))
DEST[95:64] := APPROXIMATE(1/SQRT(SRC1[95:64]))
DEST[127:96] := APPROXIMATE(1/SQRT(SRC2[127:96]))
DEST[MAXVL-1:128] (Unmodified)
```

VRSQRTPS (VEX.128 Encoded Version)

```
DEST[31:0] := APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] := APPROXIMATE(1/SQRT(SRC1[63:32]))
DEST[95:64] := APPROXIMATE(1/SQRT(SRC1[95:64]))
DEST[127:96] := APPROXIMATE(1/SQRT(SRC2[127:96]))
DEST[MAXVL-1:128] := 0
```

VRSQRTPS (VEX.256 Encoded Version)

```
DEST[31:0] := APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] := APPROXIMATE(1/SQRT(SRC1[63:32]))
DEST[95:64] := APPROXIMATE(1/SQRT(SRC1[95:64]))
DEST[127:96] := APPROXIMATE(1/SQRT(SRC2[127:96]))
DEST[159:128] := APPROXIMATE(1/SQRT(SRC2[159:128]))
DEST[191:160] := APPROXIMATE(1/SQRT(SRC2[191:160]))
DEST[223:192] := APPROXIMATE(1/SQRT(SRC2[223:192]))
DEST[255:224] := APPROXIMATE(1/SQRT(SRC2[255:224]))
```

Intel C/C++ Compiler Intrinsic Equivalent

RSQRTPS __m128 _mm_rsqrt_ps(__m128 a)
RSQRTPS __m256 _mm256_rsqrt_ps (__m256 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions," additionally:

#UD	If VEX.vvvv ≠ 1111B.

RSQRTSS—Compute Reciprocal of Square Root of Scalar Single Precision Floating-Point Value

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 52 /r RSQRTSS xmm1, xmm2/m32	RM	V/V	SSE	Computes the approximate reciprocal of the square root of the low single precision floating-point value in xmm2/m32 and stores the results in xmm1.
VEX.LIG.F3.0F.WIG 52 /r VRSQRTSS xmm1, xmm2, xmm3/m32	RVM	V/V	AVX	Computes the approximate reciprocal of the square root of the low single precision floating-point value in xmm3/m32 and stores the results in xmm1. Also, upper single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Computes an approximate reciprocal of the square root of the low single precision floating-point value in the source operand (second operand) stores the single precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a scalar single precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RSQRTSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). When a source value is a negative value (other than –0.0), a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

Operation

RSQRTSS (128-bit Legacy SSE Version)

```
DEST[31:0] := APPROXIMATE(1/SQRT(SRC2[31:0]))
DEST[MAXVL-1:32] (Unmodified)
```

VRSQRTSS (VEX.128 Encoded Version)

```
DEST[31:0] := APPROXIMATE(1/SQRT(SRC2[31:0]))
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

RSQRTSS __m128 _mm_rsqrt_ss(__m128 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-22, "Type 5 Class Exception Conditions."

RSTORSSP—Restore Saved Shadow Stack Pointer

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 01 /5 (mod!=11, /5, memory only) RSTORSSP m64	M	V/V	CET_SS	Restore SSP.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	N/A	N/A	N/A

Description

Restores SSP from the shadow-stack-restore token pointed to by m64. If the SSP restore was successful then the instruction replaces the shadow-stack-restore token with a previous-ssp token. The instruction sets the CF flag to indicate whether the SSP address recorded in the shadow-stack-restore token that was processed was 4 byte aligned, i.e., whether an alignment hole was created when the restore-shadow-stack token was pushed on this shadow stack.

Following RSTORSSP if a restore-shadow-stack token needs to be saved on the previous shadow stack, use the SAVEPREVSSP instruction.

If pushing a restore-shadow-stack token on the previous shadow stack is not required, the previous-ssp token can be popped using the INCSSPQ instruction. If the CF flag was set to indicate presence of an alignment hole, an additional INCSSPD instruction is needed to advance the SSP past the alignment hole.

Operation

```
IF CPL = 3
    IF (CR4.CET & IA32_U_CET.SH_STK_EN) = 0
        THEN #UD; FI;
ELSE
    IF (CR4.CET & IA32_S_CET.SH_STK_EN) = 0
        THEN #UD; FI;
FI;

SSP_LA = Linear_Address(mem operand)
IF SSP_LA not aligned to 8 bytes
    THEN #GP(0); FI;

previous_ssp_token = SSP | (IA32_EFER.LMA AND CS.L) | 0x02
Start Atomic Execution
restore_ssp_token = Locked shadow_stack_load 8 bytes from SSP_LA
fault = 0

IF ((restore_ssp_token & 0x03) != (IA32_EFER.LMA & CS.L))
    THEN fault = 1; FI;       (* If L flag in token does not match IA32_EFER.LMA & CS.L or bit 1 is not 0 *)

IF ((IA32_EFER.LMA AND CS.L) = 0 AND restore_ssp_token[63:32] != 0)
    THEN fault = 1; FI;       (* If compatibility/legacy mode and SSP to be restored not below 4G *)

TMP = restore_ssp_token & ~0x01
TMP = (TMP - 8)
TMP = TMP & ~0x07
IF TMP != SSP_LA
```

```
    THEN fault = 1; FI;       (* If address in token does not match the requested top of stack *)

TMP = (fault == 0) ? previous_ssp_token : restore_ssp_token
shadow_stack_store 8 bytes of TMP to SSP_LA and release lock
End Atomic Execution

IF fault == 1
   THEN #CP(RSTORSSP); FI;

SSP = SSP_LA

// Set the CF if the SSP in the restore token was 4 byte aligned, i.e., there is an alignment hole
RFLAGS.CF = (restore_ssp_token & 0x04) ? 1 : 0;
RFLAGS.ZF,PF,AF,OF,SF := 0;
```

Flags Affected

CF is set to indicate if the shadow stack pointer in the restore token was 4 byte aligned, else it is cleared. ZF, PF, AF, OF, and SF are cleared.

C/C++ Compiler Intrinsic Equivalent

RSTORSSP void _rstorssp(void *);

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	IF CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
	IF CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
#GP(0)	If linear address of memory operand not 8 byte aligned.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If destination is located in a non-writeable segment.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#CP(rstorssp)	If L bit in token does not match (IA32_EFER.LMA & CS.L).
	If address in token does not match linear address of memory operand.
	If in 32-bit or compatibility mode and the address in token is not below 4G.
#PF(fault-code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	The RSTORSSP instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The RSTORSSP instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same as protected mode exceptions.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	If CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
	If CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
#GP(0)	If linear address of memory operand not 8 byte aligned.
	If a memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#CP(rstorssp)	If L bit in token does not match (IA32_EFER.LMA & CS.L).
	If address in token does not match linear address of memory operand.
#PF(fault-code)	If a page fault occurs.

SAHF—Store AH Into Flags

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
9E	SAHF	ZO	Valid[1]	Valid	Loads SF, ZF, AF, PF, and CF from AH into the EFLAGS register.

NOTES:

1. Invalid if CPUID.80000001H:ECX.LAHF_SAHF_64 is enumerated as 0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Loads the SF, ZF, AF, PF, and CF flags of the EFLAGS register with values from the corresponding bits in the AH register (bits 7, 6, 4, 2, and 0, respectively). Bits 1, 3, and 5 of register AH are ignored; the corresponding reserved bits (1, 3, and 5) in the EFLAGS register remain as shown in the "Operation" section below.

This instruction executes as described above in compatibility mode and legacy mode. It is valid in 64-bit mode only if CPUID.80000001H:ECX.LAHF_SAHF_64[0] = 1.

Operation

```
IF IA-64 Mode
    THEN
        IF CPUID.80000001H:ECX[0] = 1;
            THEN
                RFLAGS(SF:ZF:0:AF:0:PF:1:CF) := AH;
            ELSE
                #UD;
        FI
    ELSE
        EFLAGS(SF:ZF:0:AF:0:PF:1:CF) := AH;
FI;
```

Flags Affected

The SF, ZF, AF, PF, and CF flags are loaded with values from the AH register. Bits 1, 3, and 5 of the EFLAGS register are unaffected, with the values remaining 1, 0, and 0, respectively.

Protected Mode Exceptions

None.

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

None.

Compatibility Mode Exceptions

None.

64-Bit Mode Exceptions

#UD	If CPUID.80000001H:ECX.LAHF_SAHF_64[0] = 0. If the LOCK prefix is used.

SAL/SAR/SHL/SHR—Shift

Opcode[1]	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
D0 /4	SAL r/m8[2], 1	M1	Valid	Valid	Multiply r/m8 by 2, once.
D2 /4	SAL r/m8[2], CL	MC	Valid	Valid	Multiply r/m8 by 2, CL times.
C0 /4 ib	SAL r/m8[2], imm8	MI	Valid	Valid	Multiply r/m8 by 2, imm8 times.
D1 /4	SAL r/m16, 1	M1	Valid	Valid	Multiply r/m16 by 2, once.
D3 /4	SAL r/m16, CL	MC	Valid	Valid	Multiply r/m16 by 2, CL times.
C1 /4 ib	SAL r/m16, imm8	MI	Valid	Valid	Multiply r/m16 by 2, imm8 times.
D1 /4	SAL r/m32, 1	M1	Valid	Valid	Multiply r/m32 by 2, once.
REX.W + D1 /4	SAL r/m64, 1	M1	Valid	N.E.	Multiply r/m64 by 2, once.
D3 /4	SAL r/m32, CL	MC	Valid	Valid	Multiply r/m32 by 2, CL times.
REX.W + D3 /4	SAL r/m64, CL	MC	Valid	N.E.	Multiply r/m64 by 2, CL times.
C1 /4 ib	SAL r/m32, imm8	MI	Valid	Valid	Multiply r/m32 by 2, imm8 times.
REX.W + C1 /4 ib	SAL r/m64, imm8	MI	Valid	N.E.	Multiply r/m64 by 2, imm8 times.
D0 /7	SAR r/m8[2], 1	M1	Valid	Valid	Signed divide[3] r/m8 by 2, once.
D2 /7	SAR r/m8[2], CL	MC	Valid	Valid	Signed divide[3] r/m8 by 2, CL times.
C0 /7 ib	SAR r/m8[2], imm8	MI	Valid	Valid	Signed divide[3] r/m8 by 2, imm8 times.
D1 /7	SAR r/m16,1	M1	Valid	Valid	Signed divide[3] r/m16 by 2, once.
D3 /7	SAR r/m16, CL	MC	Valid	Valid	Signed divide[3] r/m16 by 2, CL times.
C1 /7 ib	SAR r/m16, imm8	MI	Valid	Valid	Signed divide[3] r/m16 by 2, imm8 times.
D1 /7	SAR r/m32, 1	M1	Valid	Valid	Signed divide[3] r/m32 by 2, once.
REX.W + D1 /7	SAR r/m64, 1	M1	Valid	N.E.	Signed divide[3] r/m64 by 2, once.
D3 /7	SAR r/m32, CL	MC	Valid	Valid	Signed divide[3] r/m32 by 2, CL times.
REX.W + D3 /7	SAR r/m64, CL	MC	Valid	N.E.	Signed divide[3] r/m64 by 2, CL times.
C1 /7 ib	SAR r/m32, imm8	MI	Valid	Valid	Signed divide[3] r/m32 by 2, imm8 times.
REX.W + C1 /7 ib	SAR r/m64, imm8	MI	Valid	N.E.	Signed divide[3] r/m64 by 2, imm8 times
D0 /4	SHL r/m8[2], 1	M1	Valid	Valid	Multiply r/m8 by 2, once.
D2 /4	SHL r/m8[2], CL	MC	Valid	Valid	Multiply r/m8 by 2, CL times.
C0 /4 ib	SHL r/m8[2], imm8	MI	Valid	Valid	Multiply r/m8 by 2, imm8 times.
D1 /4	SHL r/m16,1	M1	Valid	Valid	Multiply r/m16 by 2, once.
D3 /4	SHL r/m16, CL	MC	Valid	Valid	Multiply r/m16 by 2, CL times.
C1 /4 ib	SHL r/m16, imm8	MI	Valid	Valid	Multiply r/m16 by 2, imm8 times.
D1 /4	SHL r/m32,1	M1	Valid	Valid	Multiply r/m32 by 2, once.
REX.W + D1 /4	SHL r/m64,1	M1	Valid	N.E.	Multiply r/m64 by 2, once.
D3 /4	SHL r/m32, CL	MC	Valid	Valid	Multiply r/m32 by 2, CL times.
REX.W + D3 /4	SHL r/m64, CL	MC	Valid	N.E.	Multiply r/m64 by 2, CL times.
C1 /4 ib	SHL r/m32, imm8	MI	Valid	Valid	Multiply r/m32 by 2, imm8 times.
REX.W + C1 /4 ib	SHL r/m64, imm8	MI	Valid	N.E.	Multiply r/m64 by 2, imm8 times.
D0 /5	SHR r/m8[2],1	M1	Valid	Valid	Unsigned divide r/m8 by 2, once.
D2 /5	SHR r/m8[2], CL	MC	Valid	Valid	Unsigned divide r/m8 by 2, CL times.
C0 /5 ib	SHR r/m8[2], imm8	MI	Valid	Valid	Unsigned divide r/m8 by 2, imm8 times.
D1 /5	SHR r/m16, 1	M1	Valid	Valid	Unsigned divide r/m16 by 2, once.

Opcode[1]	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
D3 /5	SHR r/m16, CL	MC	Valid	Valid	Unsigned divide r/m16 by 2, CL times
C1 /5 ib	SHR r/m16, imm8	MI	Valid	Valid	Unsigned divide r/m16 by 2, imm8 times.
D1 /5	SHR r/m32, 1	M1	Valid	Valid	Unsigned divide r/m32 by 2, once.
REX.W + D1 /5	SHR r/m64, 1	M1	Valid	N.E.	Unsigned divide r/m64 by 2, once.
D3 /5	SHR r/m32, CL	MC	Valid	Valid	Unsigned divide r/m32 by 2, CL times.
REX.W + D3 /5	SHR r/m64, CL	MC	Valid	N.E.	Unsigned divide r/m64 by 2, CL times.
C1 /5 ib	SHR r/m32, imm8	MI	Valid	Valid	Unsigned divide r/m32 by 2, imm8 times.
REX.W + C1 /5 ib	SHR r/m64, imm8	MI	Valid	N.E.	Unsigned divide r/m64 by 2, imm8 times.

NOTES:

1. See the IA-32 Architecture Compatibility section below.
2. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.
3. Not the same form of division as IDIV; rounding is toward negative infinity.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M1	ModRM:r/m (r, w)	1	N/A	N/A
MC	ModRM:r/m (r, w)	CL	N/A	N/A
MI	ModRM:r/m (r, w)	imm8	N/A	N/A

Description

Shifts the bits in the first operand (destination operand) to the left or right by the number of bits specified in the second operand (count operand). Bits shifted beyond the destination operand boundary are first shifted into the CF flag, then discarded. At the end of the shift operation, the CF flag contains the last bit shifted out of the destination operand.

The destination operand can be a register or a memory location. The count operand can be an immediate value or the CL register. The count is masked to 5 bits (or 6 bits with a 64-bit operand). The count range is limited to 0 to 31 (or 63 with a 64-bit operand). A special opcode encoding is provided for a count of 1.

The shift arithmetic left (SAL) and shift logical left (SHL) instructions perform the same operation; they shift the bits in the destination operand to the left (toward more significant bit locations). For each shift count, the most significant bit of the destination operand is shifted into the CF flag, and the least significant bit is cleared (see Figure 7-7 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1).

The shift arithmetic right (SAR) and shift logical right (SHR) instructions shift the bits of the destination operand to the right (toward less significant bit locations). For each shift count, the least significant bit of the destination operand is shifted into the CF flag, and the most significant bit is either set or cleared depending on the instruction type. The SHR instruction clears the most significant bit (see Figure 7-8 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1); the SAR instruction sets or clears the most significant bit to correspond to the sign (most significant bit) of the original value in the destination operand. In effect, the SAR instruction fills the empty bit position's shifted value with the sign of the unshifted value (see Figure 7-9 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1).

The SAR and SHR instructions can be used to perform signed or unsigned division, respectively, of the destination operand by powers of 2. For example, using the SAR instruction to shift a signed integer 1 bit to the right divides the value by 2.

Using the SAR instruction to perform a division operation does not produce the same result as the IDIV instruction. The quotient from the IDIV instruction is rounded toward zero, whereas the "quotient" of the SAR instruction is rounded toward negative infinity. This difference is apparent only for negative numbers. For example, when the IDIV instruction is used to divide -9 by 4, the result is -2 with a remainder of -1. If the SAR instruction is used to

shift -9 right by two bits, the result is -3 and the "remainder" is +3; however, the SAR instruction stores only the most significant bit of the remainder (in the CF flag).

The OF flag is affected only on 1-bit shifts. For left shifts, the OF flag is set to 0 if the most-significant bit of the result is the same as the CF flag (that is, the top two bits of the original operand were the same); otherwise, it is set to 1. For the SAR instruction, the OF flag is cleared for all 1-bit shifts. For the SHR instruction, the OF flag is set to the most-significant bit of the original operand.

In 64-bit mode, the instruction's default operation size is 32 bits and the mask width for CL is 5 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64-bits and sets the mask width for CL to 6 bits. See the summary chart at the beginning of this section for encoding data and limits.

IA-32 Architecture Compatibility

The 8086 does not mask the shift count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the shift count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

Operation

```
IF OperandSize = 64
    THEN
        countMASK := 3FH;
    ELSE
        countMASK := 1FH;
FI

tempCOUNT := (COUNT AND countMASK);

origDEST := DEST;
tempDEST := DEST;
WHILE (tempCOUNT ≠ 0)
DO
    IF instruction is SAL or SHL
        THEN
            CF := MSB(tempDEST);
        ELSE (* Instruction is SAR or SHR *)
            CF := LSB(tempDEST);
    FI;
    IF instruction is SAL or SHL
        THEN
            tempDEST := tempDEST * 2;
        ELSE
            IF instruction is SAR
                THEN
                    tempDEST := tempDEST / 2; (* Signed divide, rounding toward negative infinity *)
                ELSE (* Instruction is SHR *)
                    tempDEST := tempDEST / 2 ; (* Unsigned divide *)
            FI;
    FI;
    tempCOUNT := tempCOUNT – 1;
OD;

(* Determine overflow for the various instructions *)
IF (COUNT and countMASK) = 1
    THEN
        IF instruction is SAL or SHL
```

```
            THEN
                OF := MSB(tempDEST) XOR CF;
            ELSE
                IF instruction is SAR
                    THEN
                        OF := 0;
                    ELSE (* Instruction is SHR *)
                        OF := MSB(origDEST);
                FI;
        FI;
    ELSE IF (COUNT AND countMASK) = 0
        THEN
            All flags unchanged;
        ELSE (* COUNT not 1 or 0 *)
            OF := undefined;
    FI;
FI;
DEST := tempDEST;
```

Flags Affected

The CF flag contains the value of the last bit shifted out of the destination operand; it is undefined for SHL and SHR instructions where the count is greater than or equal to the size (in bits) of the destination operand. The OF flag is affected only for 1-bit shifts (see "Description" above); otherwise, it is undefined. The SF, ZF, and PF flags are set according to the result. If the count is 0, the flags are not affected. For a non-zero count, the AF flag is undefined.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

SALC—Set AL Based on Carry Flag

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
D6	SALC	ZO	N.E.[1]	Valid	Set all bits in AL to the value of CF.

NOTES:

1. In 64-bit mode, D6H encodes UDB, which raises an invalid-opcode exception.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Sets register AL to either 00H or FFH depending on the value of the carry flag (CF).

This operation is equivalent to SBB AL,AL except that it does not change the status flags.

Operation

```
IF in 64-bit mode
    THEN #UD;          // UDB
ELSE IF CF = 0
    THEN AL := 00H;
    ELSE AL := FFH;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	Encodes the UDB instruction.

SARX/SHLX/SHRX—Shift Without Affecting Flags

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.LZ.F3.0F38.W0 F7 /r SARX r32a, r/m32, r32b	RMV	V/V	BMI2	Shift r/m32 arithmetically right with count specified in r32b and load into r32a.
VEX.LZ.66.0F38.W0 F7 /r SHLX r32a, r/m32, r32b	RMV	V/V	BMI2	Shift r/m32 logically left with count specified in r32b and load into r32a.
VEX.LZ.F2.0F38.W0 F7 /r SHRX r32a, r/m32, r32b	RMV	V/V	BMI2	Shift r/m32 logically right with count specified in r32b and load into r32a.
VEX.LZ.F3.0F38.W1 F7 /r SARX r64a, r/m64, r64b	RMV	V/N.E.	BMI2	Shift r/m64 arithmetically right with count specified in r64b and load into r64a.
VEX.LZ.66.0F38.W1 F7 /r SHLX r64a, r/m64, r64b	RMV	V/N.E.	BMI2	Shift r/m64 logically left with count specified in r64b and load into r64a.
VEX.LZ.F2.0F38.W1 F7 /r SHRX r64a, r/m64, r64b	RMV	V/N.E.	BMI2	Shift r/m64 logically right with count specified in r64b and load into r64a.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (w)	ModRM:r/m (r)	VEX.vvvv (r)	N/A

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;

DO WHILE (COUNT ≠ 0)
    IF instruction is SHLX
        THEN
            TEMP := TEMP *2;
```

```
        ELSE IF instruction is SHRX
            THEN
                TEMP := TEMP /2; //unsigned divide
        ELSE            // SARX
                TEMP := TEMP /2; // signed divide, round toward negative infinity
    FI;
    COUNT := COUNT - 1;
OD
DEST := TEMP;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-29, "Type 13 Class Exception Conditions."

SAVEPREVSSP—Save Previous Shadow Stack Pointer

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 01 EA SAVEPREVSSP	ZO	V/V	CET_SS	Save a restore-shadow-stack token on previous shadow stack.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Push a restore-shadow-stack token on the previous shadow stack at the next 8 byte aligned boundary. The previous SSP is obtained from the previous-ssp token at the top of the current shadow stack.

Operation

```
IF CPL = 3
    IF (CR4.CET & IA32_U_CET.SH_STK_EN) = 0
        THEN #UD; FI;
ELSE
    IF (CR4.CET & IA32_S_CET.SH_STK_EN) = 0
        THEN #UD; FI;
FI;

IF SSP not aligned to 8 bytes
    THEN #GP(0); FI;
(* Pop the "previous-ssp" token from current shadow stack *)
previous_ssp_token = ShadowStackPop8B(SSP)

(* If the CF flag indicates there was a alignment hole on current shadow stack then pop that alignment hole *)
(* Note that the alignment hole must be zero and can be present only when in legacy/compatibility mode *)
IF RFLAGS.CF == 1 AND (IA32_EFER.LMA AND CS.L)
    #GP(0)
FI;
IF RFLAGS.CF == 1
    must_be_zero = ShadowStackPop4B(SSP)
    IF must_be_zero != 0 THEN #GP(0)
FI;

(* Previous SSP token must have the bit 1 set *)
IF ((previous_ssp_token & 0x02) == 0)
    THEN #GP(0); (* bit 1 was 0 *)

IF ((IA32_EFER.LMA AND CS.L) = 0 AND previous_ssp_token [63:32] != 0)
THEN #GP(0); FI; (* If compatibility/legacy mode and SSP not in 4G *)

(* Save Prev SSP from previous_ssp_token to the old shadow stack at next 8 byte aligned address *)
old_SSP = previous_ssp_token & ~0x03
temp := (old_SSP | (IA32_EFER.LMA & CS.L));
Shadow_stack_store 4 bytes of 0 to (old_SSP - 4)
old_SSP := old_SSP & ~0x07;
```

Shadow_stack_store 8 bytes of temp to (old_SSP - 8)

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

SAVEPREVSSP void _saveprevssp(void);

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	IF CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
	IF CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
#GP(0)	If SSP not 8 byte aligned.
	If alignment hole on shadow stack is not 0.
	If bit 1 of the previous-ssp token is not set to 1.
	If in 32-bit/compatibility mode and SSP recorded in previous-ssp token is beyond 4G.
#PF(fault-code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	The SAVEPREVSSP instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The SAVEPREVSSP instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same as protected mode exceptions.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	If CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
	If CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
#GP(0)	If SSP not 8 byte aligned.
	If carry flag is set.
	If bit 1 of the previous-ssp token is not set to 1.
#PF(fault-code)	If a page fault occurs.

SBB—Integer Subtraction With Borrow

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
1C ib	SBB AL, imm8	I	Valid	Valid	Subtract with borrow imm8 from AL.
1D iw	SBB AX, imm16	I	Valid	Valid	Subtract with borrow imm16 from AX.
1D id	SBB EAX, imm32	I	Valid	Valid	Subtract with borrow imm32 from EAX.
REX.W + 1D id	SBB RAX, imm32	I	Valid	N.E.	Subtract with borrow sign-extended imm.32 to 64-bits from RAX.
80 /3 ib	SBB r/m8[1], imm8	MI	Valid	Valid	Subtract with borrow imm8 from r/m8.
81 /3 iw	SBB r/m16, imm16	MI	Valid	Valid	Subtract with borrow imm16 from r/m16.
81 /3 id	SBB r/m32, imm32	MI	Valid	Valid	Subtract with borrow imm32 from r/m32.
REX.W + 81 /3 id	SBB r/m64, imm32	MI	Valid	N.E.	Subtract with borrow sign-extended imm32 to 64-bits from r/m64.
83 /3 ib	SBB r/m16, imm8	MI	Valid	Valid	Subtract with borrow sign-extended imm8 from r/m16.
83 /3 ib	SBB r/m32, imm8	MI	Valid	Valid	Subtract with borrow sign-extended imm8 from r/m32.
REX.W + 83 /3 ib	SBB r/m64, imm8	MI	Valid	N.E.	Subtract with borrow sign-extended imm8 from r/m64.
18 /r	SBB r/m8[1], r8[1]	MR	Valid	Valid	Subtract with borrow r8 from r/m8.
19 /r	SBB r/m16, r16	MR	Valid	Valid	Subtract with borrow r16 from r/m16.
19 /r	SBB r/m32, r32	MR	Valid	Valid	Subtract with borrow r32 from r/m32.
REX.W + 19 /r	SBB r/m64, r64	MR	Valid	N.E.	Subtract with borrow r64 from r/m64.
1A /r	SBB r8[1], r/m8[1]	RM	Valid	Valid	Subtract with borrow r/m8 from r8.
1B /r	SBB r16, r/m16	RM	Valid	Valid	Subtract with borrow r/m16 from r16.
1B /r	SBB r32, r/m32	RM	Valid	Valid	Subtract with borrow r/m32 from r32.
REX.W + 1B /r	SBB r64, r/m64	RM	Valid	N.E.	Subtract with borrow r/m64 from r64.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	imm8/16/32	N/A	N/A
MI	ModRM:r/m (w)	imm8/16/32	N/A	N/A
MR	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Adds the source operand (second operand) and the carry (CF) flag, and subtracts the result from the destination operand (first operand). The result of the subtraction is stored in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) The state of the CF flag represents a borrow from a previous subtraction.

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The SBB instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a borrow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

The SBB instruction is usually executed as part of a multibyte or multiword subtraction in which a SUB instruction is followed by a SBB instruction.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST := (DEST – (SRC + CF));
```

Intel C/C++ Compiler Intrinsic Equivalent

SBB extern unsigned char _subborrow_u8(unsigned char c_in, unsigned char src1, unsigned char src2, unsigned char *diff_out);
SBB extern unsigned char _subborrow_u16(unsigned char c_in, unsigned short src1, unsigned short src2, unsigned short *diff_out);
SBB extern unsigned char _subborrow_u32(unsigned char c_in, unsigned int src1, unsigned char int, unsigned int *diff_out);
SBB extern unsigned char _subborrow_u64(unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *diff_out);

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

SCAS/SCASB/SCASW/SCASD—Scan String

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
AE	SCAS m8	ZO	Valid	Valid	Compare AL with byte at ES:(E)DI or RDI, then set status flags.[1]
AF	SCAS m16	ZO	Valid	Valid	Compare AX with word at ES:(E)DI or RDI, then set status flags.[1]
AF	SCAS m32	ZO	Valid	Valid	Compare EAX with doubleword at ES(E)DI or RDI then set status flags.[1]
REX.W + AF	SCAS m64	ZO	Valid	N.E.	Compare RAX with quadword at RDI or EDI then set status flags.
AE	SCASB	ZO	Valid	Valid	Compare AL with byte at ES:(E)DI or RDI then set status flags.[1]
AF	SCASW	ZO	Valid	Valid	Compare AX with word at ES:(E)DI or RDI then set status flags.[1]
AF	SCASD	ZO	Valid	Valid	Compare EAX with doubleword at ES:(E)DI or RDI then set status flags.[1]
REX.W + AF	SCASQ	ZO	Valid	N.E.	Compare RAX with quadword at RDI or EDI then set status flags.

NOTES:

1. In 64-bit mode, only 64-bit (RDI) and 32-bit (EDI) address sizes are supported. In non-64-bit mode, only 32-bit (EDI) and 16-bit (DI) address sizes are supported.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

In non-64-bit modes and in default 64-bit mode: this instruction compares a byte, word, doubleword or quadword specified using a memory operand with the value in AL, AX, or EAX. It then sets status flags in EFLAGS recording the results. The memory operand address is read from ES:(E)DI register (depending on the address-size attribute of the instruction and the current operational mode). Note that ES cannot be overridden with a segment override prefix.

At the assembly-code level, two forms of this instruction are allowed. The explicit-operand form and the no-operands form. The explicit-operand form (specified using the SCAS mnemonic) allows a memory operand to be specified explicitly. The memory operand must be a symbol that indicates the size and location of the operand value. The register operand is then automatically selected to match the size of the memory operand (AL register for byte comparisons, AX for word comparisons, EAX for doubleword comparisons). The explicit-operand form is provided to allow documentation. Note that the documentation provided by this form can be misleading. That is, the memory operand symbol must specify the correct type (size) of the operand (byte, word, or doubleword) but it does not have to specify the correct location. The location is always specified by ES:(E)DI.

The no-operands form of the instruction uses a short form of SCAS. Again, ES:(E)DI is assumed to be the memory operand and AL, AX, or EAX is assumed to be the register operand. The size of operands is selected by the mnemonic: SCASB (byte comparison), SCASW (word comparison), or SCASD (doubleword comparison).

After the comparison, the (E)DI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. If the DF flag is 0, the (E)DI register is incremented; if the DF flag is 1, the (E)DI register is decremented. The register is incremented or decremented by 1 for byte operations, by 2 for word operations, and by 4 for doubleword operations.

SCAS, SCASB, SCASW, SCASD, and SCASQ can be preceded by the REPE prefix or the REPNE prefix for block comparisons of ECX bytes, words, doublewords, or quadwords. Often, however, these instructions will be used in a LOOP construct that takes some action based on the setting of status flags. See "REPE/REPZ—Repeat String Operation While Zero (Prefix)" and "REPNE/REPNZ—Repeat String Operation While Not Zero (Prefix)" in this chapter for a description of the REPE and REPNE prefixes.

In 64-bit mode, the instruction's default address size is 64-bits, 32-bit address size is supported using the prefix 67H. Using a REX prefix in the form of REX.W promotes operation on doubleword operand to 64 bits. The 64-bit no-operand mnemonic is SCASQ. Address of the memory operand is specified in either RDI or EDI, and AL/AX/EAX/RAX may be used as the register operand. After a comparison, the destination register is incremented or decremented by the current operand size (depending on the value of the DF flag). See the summary chart at the beginning of this section for encoding data and limits.

Operation

Non-64-bit Mode:

```
IF (Byte comparison)
    THEN
        temp := AL – SRC;
        SetStatusFlags(temp);
            THEN IF DF = 0
                THEN (E)DI := (E)DI + 1;
                ELSE (E)DI := (E)DI – 1; FI;
    ELSE IF (Word comparison)
        THEN
            temp := AX – SRC;
            SetStatusFlags(temp);
            IF DF = 0
                THEN (E)DI := (E)DI + 2;
                ELSE (E)DI := (E)DI – 2; FI;
        FI;
    ELSE IF (Doubleword comparison)
        THEN
            temp := EAX – SRC;
            SetStatusFlags(temp);
            IF DF = 0
                THEN (E)DI := (E)DI + 4;
                ELSE (E)DI := (E)DI – 4; FI;
        FI;
FI;
```

64-bit Mode:

```
IF (Byte comparison)
    THEN
        temp := AL – SRC;
        SetStatusFlags(temp);
            THEN IF DF = 0
                THEN (R|E)DI := (R|E)DI + 1;
                ELSE (R|E)DI := (R|E)DI – 1; FI;
    ELSE IF (Word comparison)
        THEN
            temp := AX – SRC;
            SetStatusFlags(temp);
            IF DF = 0
                THEN (R|E)DI := (R|E)DI + 2;
                ELSE (R|E)DI := (R|E)DI – 2; FI;
```

```
        FI;
    ELSE IF (Doubleword comparison)
        THEN
            temp := EAX – SRC;
            SetStatusFlags(temp);
            IF DF = 0
                THEN (R|E)DI := (R|E)DI + 4;
                ELSE (R|E)DI := (R|E)DI – 4; FI;
        FI;
    ELSE IF (Quadword comparison using REX.W )
        THEN
            temp := RAX — SRC;
            SetStatusFlags(temp);
            IF DF = 0
                THEN (R|E)DI := (R|E)DI + 8;
                ELSE (R|E)DI := (R|E)DI – 8;
            FI;
    FI;
FI;
```

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are set according to the temporary result of the comparison.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the limit of the ES segment.
	If the ES register contains a NULL segment selector.
	If an illegal memory operand effective address in the ES segment is given.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

SENDUIPI—Send User Interprocessor Interrupt

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F C7 /6 SENDUIPI reg	A	V/I	UINTR	Send interprocessor user interrupt.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r)	N/A	N/A	N/A

Description

The SENDUIPI instruction sends the user interprocessor interrupt (IPI) indicated by its register operand. (The operand always has 64 bits; operand-size overrides such as the prefix 66 are ignored.)

SENDUIPI uses a data structure called the user-interrupt target table (UITT). This table is located at the linear address UITTADDR (in the IA32_UINTR_TT MSR); it comprises UITTSZ+1 16-byte entries, where UITTSZ = IA32_UINT_MISC[31:0]. SENDUIPI uses the UITT entry (UITTE) indexed by the instruction's register operand. Each UITTE has the following format:

- Bit 0: V, a valid bit.
- Bits 7:1 are reserved and must be 0.
- Bits 15:8: UV, the user-interrupt vector (in the range 0–63, so bits 15:14 must be 0).
- Bits 63:16 are reserved.
- Bits 127:64: UPIDADDR, the linear address of a user posted-interrupt descriptor (UPID). (UPIDADDR is 64-byte aligned, so bits 69:64 of each UITTE must be 0.)

Each UPID has the following format (fields and bits not referenced are reserved):

- Bit 0 (ON) indicates an outstanding notification. If this bit is set, there is a notification outstanding for one or more user interrupts in PIR.
- Bit 1 (SN) indicates that notifications should be suppressed. If this bit is set, agents (including SENDUIPI) should not send notifications when posting user interrupts in this descriptor.
- Bits 23:16 (NV) contain the notification vector. This is used by agents sending user-interrupt notifications (including SENDUIPI).
- Bits 63:32 (NDST) contain the notification destination. This is the target physical APIC ID (in xAPIC mode, bits 47:40 are the 8-bit APIC ID; in x2APIC mode, the entire field forms the 32-bit APIC ID).
- Bits 127:64 (PIF) contain posted-interrupt requests. There is one bit for each user-interrupt vector. There is a user-interrupt request for a vector if the corresponding bit is 1.

Although SENDUIPI may be executed at any privilege level, all of the instruction’s memory accesses (to a UITTE and a UPID) are performed with supervisor privilege.

SENDUIPI sends a user interrupt by posting a user interrupt with vector V in the UPID referenced by UPIDADDR and then sending, as an ordinary IPI, any notification interrupt specified in that UPID.

Operation

```
IF reg > UITTSZ;
    THEN #GP(0);
FI;
read tempUITTE from 16 bytes at UITTADDR+ (reg « 4);
IF tempUITTE.V = 0 or tempUITTE sets any reserved bit
    THEN #GP(0);
FI;
```

```
read tempUPID from 16 bytes at tempUITTE.UPIDADDR;// under lock
IF tempUPID sets any reserved bits or bits that must be zero
    THEN #GP(0); // release lock
FI;
tempUPID.PIR[tempUITTE.UV] := 1;
IF tempUPID.SN = tempUPID.ON = 0
    THEN
        tempUPID.ON := 1;
        sendNotify := 1;
    ELSE sendNotify := 0;
FI;
write tempUPID to 16 bytes at tempUITTE.UPIDADDR;// release lock
IF sendNotify = 1
    THEN
        IF local APIC is in x2APIC mode
            THEN send ordinary IPI with vector tempUPID.NV
                to 32-bit physical APIC ID tempUPID.NDST;
            ELSE send ordinary IPI with vector tempUPID.NV
                to 8-bit physical APIC ID tempUPID.NDST[15:8];
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#UD	The SENDUIPI instruction is not recognized in protected mode.

Real-Address Mode Exceptions

#UD	The SENDUIPI instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The SENDUIPI instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	The SENDUIPI instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If executed inside an enclave.
	If CR4.UINTR = 0.
	If IA32_UINTR_TT[0] = 0.
	If CPUID.07H.00H:EDX.UINTR[5] = 0.
#PF	If a page fault occurs.
#GP	If the value of the register operand exceeds UITTSZ.
	If the selected UITTE is not valid or sets any reserved bits.
	If the selected UPID sets any reserved bits.
	If there is an attempt to access memory using a linear address that is not canonical relative to the current paging mode.

SERIALIZE—Serialize Instruction Execution

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 E8 SERIALIZE	ZO	V/V	SERIALIZE	Serialize instruction fetch and execution.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A	N/A

Description

Serializes instruction execution. Before the next instruction is fetched and executed, the SERIALIZE instruction ensures that all modifications to flags, registers, and memory by previous instructions are completed, draining all buffered writes to memory. This instruction is also a serializing instruction as defined in the section "Serializing Instructions" in Chapter 11 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

SERIALIZE does not modify registers, arithmetic flags, or memory.

Operation

```
Wait_On_Fetch_And_Execution_Of_Next_Instruction_Until(preceding_instructions_complete_and_preceding_stores_globally_visible);
```

Intel C/C++ Compiler Intrinsic Equivalent

SERIALIZE void _serialize(void);

SIMD Floating-Point Exceptions

None.

Other Exceptions

#UD	If the LOCK prefix is used. If CPUID.07H.00H:EDX.SERIALIZE[14] = 0.

SET*cc*—Set Byte on Condition

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 97	SETA r/m8[1]	M	Valid	Valid	Set byte if above (CF=0 and ZF=0).
0F 93	SETAE r/m8[1]	M	Valid	Valid	Set byte if above or equal (CF=0).
0F 92	SETB r/m8[1]	M	Valid	Valid	Set byte if below (CF=1).
0F 96	SETBE r/m8[1]	M	Valid	Valid	Set byte if below or equal (CF=1 or ZF=1).
0F 92	SETC r/m8[1]	M	Valid	Valid	Set byte if carry (CF=1).
0F 94	SETE r/m8[1]	M	Valid	Valid	Set byte if equal (ZF=1).
0F 9F	SETG r/m8[1]	M	Valid	Valid	Set byte if greater (ZF=0 and SF=OF).
0F 9D	SETGE r/m8[1]	M	Valid	Valid	Set byte if greater or equal (SF=OF).
0F 9C	SETL r/m8[1]	M	Valid	Valid	Set byte if less (SF≠ OF).
0F 9E	SETLE r/m8[1]	M	Valid	Valid	Set byte if less or equal (ZF=1 or SF≠ OF).
0F 96	SETNA r/m8[1]	M	Valid	Valid	Set byte if not above (CF=1 or ZF=1).
0F 92	SETNAE r/m8[1]	M	Valid	Valid	Set byte if not above or equal (CF=1).
0F 93	SETNB r/m8[1]	M	Valid	Valid	Set byte if not below (CF=0).
0F 97	SETNBE r/m8[1]	M	Valid	Valid	Set byte if not below or equal (CF=0 and ZF=0).
0F 93	SETNC r/m8[1]	M	Valid	Valid	Set byte if not carry (CF=0).
0F 95	SETNE r/m8[1]	M	Valid	Valid	Set byte if not equal (ZF=0).
0F 9E	SETNG r/m8[1]	M	Valid	Valid	Set byte if not greater (ZF=1 or SF≠ OF)
0F 9C	SETNGE r/m8[1]	M	Valid	Valid	Set byte if not greater or equal (SF≠ OF).
0F 9D	SETNL r/m8[1]	M	Valid	Valid	Set byte if not less (SF=OF).
0F 9F	SETNLE r/m8[1]	M	Valid	Valid	Set byte if not less or equal (ZF=0 and SF=OF).
0F 91	SETNO r/m8[1]	M	Valid	Valid	Set byte if not overflow (OF=0).
0F 9B	SETNP r/m8[1]	M	Valid	Valid	Set byte if not parity (PF=0).
0F 99	SETNS r/m8[1]	M	Valid	Valid	Set byte if not sign (SF=0).
0F 95	SETNZ r/m8[1]	M	Valid	Valid	Set byte if not zero (ZF=0).
0F 90	SETO r/m8[1]	M	Valid	Valid	Set byte if overflow (OF=1)
0F 9A	SETP r/m8[1]	M	Valid	Valid	Set byte if parity (PF=1).
0F 9A	SETPE r/m8[1]	M	Valid	Valid	Set byte if parity even (PF=1).
0F 9B	SETPO r/m8[1]	M	Valid	Valid	Set byte if parity odd (PF=0).
0F 98	SETS r/m8[1]	M	Valid	Valid	Set byte if sign (SF=1).
0F 94	SETZ r/m8[1]	M	Valid	Valid	Set byte if zero (ZF=1).

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	N/A	N/A	N/A

Description

Sets the destination operand to 0 or 1 depending on the settings of the status flags (CF, SF, OF, ZF, and PF) in the EFLAGS register. The destination operand points to a byte register or a byte in memory. The condition code suffix (*cc*) indicates the condition being tested for.

The terms "above" and "below" are associated with the CF flag and refer to the relationship between two unsigned integer values. The terms "greater" and "less" are associated with the SF and OF flags and refer to the relationship between two signed integer values.

Many of the SET*cc* instruction opcodes have alternate mnemonics. For example, SETG (set byte if greater) and SETNLE (set if not less or equal) have the same opcode and test for the same condition: ZF equals 0 and SF equals OF. These alternate mnemonics are provided to make code more intelligible. Appendix B, "EFLAGS Condition Codes," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, shows the alternate mnemonics for various test conditions.

Some languages represent a logical one as an integer with all bits set. This representation can be obtained by choosing the logically opposite condition for the SET*cc* instruction, then decrementing the result. For example, to test for overflow, use the SETNO instruction, then decrement the result.

The reg field of the ModR/M byte is not used for the SETCC instruction and those opcode bits are ignored by the processor.

In IA-64 mode, the operand size is fixed at 8 bits. Use of REX prefix enable uniform addressing to additional byte registers. Otherwise, this instruction's operation is the same as in legacy mode and compatibility mode.

Operation

```
IF condition
    THEN DEST := 1;
    ELSE DEST := 0;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.

#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

SETSSBSY—Mark Shadow Stack Busy

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 01 E8 SETSSBSY	ZO	V/V	CET_SS	Set busy flag in supervisor shadow stack token reference by IA32_PL0_SSP.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

The SETSSBSY instruction verifies the presence of a non-busy supervisor shadow stack token at the address in the IA32_PL0_SSP MSR and marks it busy. Following successful execution of the instruction, the SSP is set to the value of the IA32_PL0_SSP MSR.

This instruction cannot be executed when FRED transitions are enabled. FRED transitions do not use supervisor shadow stack tokens.

Operation

```
IF CR4.CET = 0 OR CR4.FRED = 1
    THEN #UD; FI;
IF IA32_S_CET.SH_STK_EN = 0
    THEN #UD; FI;
IF CPL > 0
    THEN GP(0); FI;

SSP_LA = IA32_PL0_SSP
If SSP_LA not aligned to 8 bytes
    THEN #GP(0); FI;

expected_token_value = SSP_LA                    (* busy bit must not be set *)
new_token_value      = SSP_LA | BUSY_BIT         (* set busy bit; bit position 0 *)
IF shadow_stack_lock_cmpxchg8B(SSP_LA, new_token_value, expected_token_value) != expected_token_value
    THEN #CP(SETSSBSY); FI;
SSP = SSP_LA
```

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

SETSSBSYvoid _setssbsy(void);

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	IF IA32_S_CET.SH_STK_EN = 0.
#GP(0)	If IA32_PL0_SSP not aligned to 8 bytes.
	If CPL is not 0.

#CP(setssbsy)	If busy bit in token is set. If the address in token is not below 4 GBytes.
#PF(fault-code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	The SETSSBSY instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The SETSSBSY instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used. If CR4.CET = 0. IF IA32_S_CET.SH_STK_EN = 0. If CR4.FRED = 1.
#GP(0)	If IA32_PL0_SSP not aligned to 8 bytes. If CPL is not 0.
#CP(setssbsy)	If busy bit in token is set. If the address in token is not below 4 GBytes.
#PF(fault-code)	If a page fault occurs.

64-Bit Mode Exceptions

Same as compatibility mode exceptions (except there is no check on the token address).

SFENCE—Store Fence

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F AE F8	SFENCE	ZO	Valid	Valid	Serializes store operations.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Orders processor execution relative to all memory stores prior to the SFENCE instruction. The processor ensures that every store prior to SFENCE is globally visible before any store after SFENCE becomes globally visible. The SFENCE instruction is ordered with respect to memory stores, other SFENCE instructions, MFENCE instructions, and any serializing instructions (such as the CPUID instruction). It is not ordered with respect to memory loads or the LFENCE instruction.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, write-combining, and write-collapsing. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The SFENCE instruction provides a performance-efficient way of ensuring store ordering between routines that produce weakly-ordered results and routines that consume this data.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Specification of the instruction's opcode above indicates a ModR/M byte of F8. For this instruction, the processor ignores the r/m field of the ModR/M byte. Thus, SFENCE is encoded by any opcode of the form 0F AE Fx, where x is in the range 8-F.

Operation

```
Wait_On_Following_Stores_Until(preceding_stores_globally_visible);
```

Intel C/C++ Compiler Intrinsic Equivalent

void _mm_sfence(void)

Exceptions (All Operating Modes)

#UD	If CPUID.01H:EDX.SSE[25] = 0.
	If the LOCK prefix is used.

SGDT—Store Global Descriptor Table Register

Opcode[1]	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 /0	SGDT *m*	M	Valid	Valid	Store GDTR to *m*.

NOTES:

1. See the IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	N/A	N/A	N/A

Description

Stores the content of the global descriptor table register (GDTR) in the destination operand. The destination operand specifies a memory location.

In legacy or compatibility mode, the destination operand is a 6-byte memory location. If the operand-size attribute is 16 or 32 bits, the 16-bit limit field of the register is stored in the low 2 bytes of the memory location and the 32-bit base address is stored in the high 4 bytes.

In 64-bit mode, the operand size is fixed at 8+2 bytes. The instruction stores an 8-byte base and a 2-byte limit.

SGDT is useful only by operating-system software. However, it can be used in application programs without causing an exception to be generated if CR4.UMIP = 0. See "LGDT/LIDT—Load Global/Interrupt Descriptor Table Register" in Chapter 3, Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, for information on loading the GDTR and IDTR.

IA-32 Architecture Compatibility

The 16-bit form of the SGDT is compatible with the Intel 286 processor if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; processor generations later than the Intel 286 processor fill these bits with 0s.

Operation

```
IF instruction is SGDT
        IF OperandSize =16 or OperandSize = 32 (* Legacy or Compatibility Mode *)
            THEN
                DEST[0:15] := GDTR(Limit);
                DEST[16:47] := GDTR(Base); (* Full 32-bit base address stored *)
                FI;
            ELSE (* 64-bit Mode *)
                DEST[0:15] := GDTR(Limit);
                DEST[16:79] := GDTR(Base); (* Full 64-bit base address stored *)
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If CR4.UMIP = 1 and CPL > 0.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If CR4.UMIP = 1.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#UD	If the LOCK prefix is used.
#GP(0)	If the memory address is in a non-canonical form.
	If CR4.UMIP = 1 and CPL > 0.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.

SHA1MSG1—Perform an Intermediate Calculation for the Next Four SHA1 Message Dwords

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 C9 /r SHA1MSG1 xmm1, xmm2/m128	RM	V/V	SHA	Performs an intermediate calculation for the next four SHA1 message dwords using previous message dwords from xmm1 and xmm2/m128, storing the result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A

Description

The SHA1MSG1 instruction is one of two SHA1 message scheduling instructions. The instruction performs an intermediate calculation for the next four SHA1 message dwords.

Operation

SHA1MSG1

```
W0 := SRC1[127:96] ;
W1 := SRC1[95:64] ;
W2 := SRC1[63: 32] ;
W3 := SRC1[31: 0] ;
W4 := SRC2[127:96] ;
W5 := SRC2[95:64] ;

DEST[127:96] := W2 XOR W0;
DEST[95:64] := W3 XOR W1;
DEST[63:32] := W4 XOR W2;
DEST[31:0] := W5 XOR W3;
```

Intel C/C++ Compiler Intrinsic Equivalent

SHA1MSG1 __m128i _mm_sha1msg1_epu32(__m128i, __m128i);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions."

SHA1MSG2—Perform a Final Calculation for the Next Four SHA1 Message Dwords

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 CA /r SHA1MSG2 xmm1, xmm2/m128	RM	V/V	SHA	Performs the final calculation for the next four SHA1 message dwords using intermediate results from xmm1 and the previous message dwords from xmm2/m128, storing the result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A

Description

The SHA1MSG2 instruction is one of two SHA1 message scheduling instructions. The instruction performs the final calculation to derive the next four SHA1 message dwords.

Operation

SHA1MSG2

```
W13 := SRC2[95:64] ;
W14 := SRC2[63: 32] ;
W15 := SRC2[31: 0] ;
W16 := (SRC1[127:96] XOR W13 ) ROL 1;
W17 := (SRC1[95:64] XOR W14) ROL 1;
W18 := (SRC1[63: 32] XOR W15) ROL 1;
W19 := (SRC1[31: 0] XOR W16) ROL 1;

DEST[127:96] := W16;
DEST[95:64] := W17;
DEST[63:32] := W18;
DEST[31:0] := W19;
```

Intel C/C++ Compiler Intrinsic Equivalent

SHA1MSG2 __m128i _mm_sha1msg2_epu32(__m128i, __m128i);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions."

SHA1NEXTE—Calculate SHA1 State Variable E After Four Rounds

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 C8 /r SHA1NEXTE xmm1, xmm2/m128	RM	V/V	SHA	Calculates SHA1 state variable E after four rounds of operation from the current SHA1 state variable A in xmm1. The calculated value of the SHA1 state variable E is added to the scheduled dwords in xmm2/m128, and stored with some of the scheduled dwords in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A

Description

The SHA1NEXTE calculates the SHA1 state variable E after four rounds of operation from the current SHA1 state variable A in the destination operand. The calculated value of the SHA1 state variable E is added to the source operand, which contains the scheduled dwords.

Operation

SHA1NEXTE

```
TMP := (SRC1[127:96] ROL 30);

DEST[127:96] := SRC2[127:96] + TMP;
DEST[95:64] := SRC2[95:64];
DEST[63:32] := SRC2[63:32];
DEST[31:0] := SRC2[31:0];
```

Intel C/C++ Compiler Intrinsic Equivalent

SHA1NEXTE __m128i _mm_sha1nexte_epu32(__m128i, __m128i);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions."

SHA1RNDS4—Perform Four Rounds of SHA1 Operation

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 3A CC /r ib SHA1RNDS4 xmm1, xmm2/m128, imm8	RMI	V/V	SHA	Performs four rounds of SHA1 operation operating on SHA1 state (A,B,C,D) from xmm1, with a pre-computed sum of the next 4 round message dwords and state variable E from xmm2/m128. The immediate byte controls logic functions and round constants.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8

Description

The SHA1RNDS4 instruction performs four rounds of SHA1 operation using an initial SHA1 state (A,B,C,D) from the first operand (which is a source operand and the destination operand) and some pre-computed sum of the next 4 round message dwords, and state variable E from the second operand (a source operand). The updated SHA1 state (A,B,C,D) after four rounds of processing is stored in the destination operand.

Operation

SHA1RNDS4

The function f() and Constant K are dependent on the value of the immediate.

```
IF ( imm8[1:0] = 0 )
    THEN f() := f0(), K := K0;
ELSE IF ( imm8[1:0] = 1 )
    THEN f() := f1(), K := K1;
ELSE IF ( imm8[1:0] = 2 )
    THEN f() := f2(), K := K2;
ELSE IF ( imm8[1:0] = 3 )
    THEN f() := f3(), K := K3;
FI;

A := SRC1[127:96];
B := SRC1[95:64];
C := SRC1[63:32];
D := SRC1[31:0];
W0E := SRC2[127:96];
W1 := SRC2[95:64];
W2 := SRC2[63:32];
W3 := SRC2[31:0];

Round i = 0 operation:
A_1 := f (B, C, D) + (A ROL 5) +W0E +K;
B_1 := A;
C_1 := B ROL 30;
D_1 := C;
E_1 := D;

FOR i = 1 to 3
    A_(i +1) := f (B_i, C_i, D_i) + (A_i ROL 5) +Wi+ E_i +K;
```

```
    B_(i +1) := A_i;
    C_(i +1) := B_i ROL 30;
    D_(i +1) := C_i;
    E_(i +1) := D_i;
ENDFOR

DEST[127:96] := A_4;
DEST[95:64] := B_4;
DEST[63:32] := C_4;
DEST[31:0] := D_4;
```

Intel C/C++ Compiler Intrinsic Equivalent

SHA1RNDS4 __m128i _mm_sha1rnds4_epu32(__m128i, __m128i, const int);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions."

SHA256MSG1—Perform an Intermediate Calculation for the Next Four SHA256 Message Dwords

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 CC /r SHA256MSG1 xmm1, xmm2/m128	RM	V/V	SHA	Performs an intermediate calculation for the next four SHA256 message dwords using previous message dwords from xmm1 and xmm2/m128, storing the result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A

Description

The SHA256MSG1 instruction is one of two SHA256 message scheduling instructions. The instruction performs an intermediate calculation for the next four SHA256 message dwords.

Operation

SHA256MSG1

W4 := SRC2[31: 0] ;
W3 := SRC1[127:96] ;
W2 := SRC1[95:64] ;
W1 := SRC1[63: 32] ;
W0 := SRC1[31: 0] ;

DEST[127:96] := W3 + σ_0(W4);
DEST[95:64] := W2 + σ_0(W3);
DEST[63:32] := W1 + σ_0(W2);
DEST[31:0] := W0 + σ_0(W1);

Intel C/C++ Compiler Intrinsic Equivalent

SHA256MSG1 __m128i _mm_sha256msg1_epu32(__m128i, __m128i);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions."

SHA256MSG2—Perform a Final Calculation for the Next Four SHA256 Message Dwords

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 CD /r SHA256MSG2 xmm1, xmm2/m128	RM	V/V	SHA	Performs the final calculation for the next four SHA256 message dwords using previous message dwords from xmm1 and xmm2/m128, storing the result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A

Description

The SHA256MSG2 instruction is one of two SHA2 message scheduling instructions. The instruction performs the final calculation for the next four SHA256 message dwords.

Operation

SHA256MSG2

W14 := SRC2[95:64] ;
W15 := SRC2[127:96] ;
W16 := SRC1[31: 0] + σ_1(W14) ;
W17 := SRC1[63: 32] + σ_1(W15) ;
W18 := SRC1[95: 64] + σ_1(W16) ;
W19 := SRC1[127: 96] + σ_1(W17) ;

DEST[127:96] := W19 ;
DEST[95:64] := W18 ;
DEST[63:32] := W17 ;
DEST[31:0] := W16;

Intel C/C++ Compiler Intrinsic Equivalent

SHA256MSG2 __m128i _mm_sha256msg2_epu32(__m128i, __m128i);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions."

SHA256RNDS2—Perform Two Rounds of SHA256 Operation

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 CB /r SHA256RNDS2 xmm1, xmm2/m128, <XMM0>	RMI	V/V	SHA	Perform 2 rounds of SHA256 operation using an initial SHA256 state (C,D,G,H) from xmm1, an initial SHA256 state (A,B,E,F) from xmm2/m128, and a pre-computed sum of the next 2 round message dwords and the corresponding round constants from the implicit operand XMM0, storing the updated SHA256 state (A,B,E,F) result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Implicit XMM0 (r)

Description

The SHA256RNDS2 instruction performs 2 rounds of SHA256 operation using an initial SHA256 state (C,D,G,H) from the first operand, an initial SHA256 state (A,B,E,F) from the second operand, and a pre-computed sum of the next 2 round message dwords and the corresponding round constants from the implicit operand xmm0. Note that only the two lower dwords of XMM0 are used by the instruction.

The updated SHA256 state (A,B,E,F) is written to the first operand, and the second operand can be used as the updated state (C,D,G,H) in later rounds.

Operation

SHA256RNDS2

```
A_0 := SRC2[127:96];
B_0 := SRC2[95:64];
C_0 := SRC1[127:96];
D_0 := SRC1[95:64];
E_0 := SRC2[63:32];
F_0 := SRC2[31:0];
G_0 := SRC1[63:32];
H_0 := SRC1[31:0];
WK_0 := XMM0[31: 0];
WK_1 := XMM0[63: 32];

FOR i = 0 to 1
    A_(i +1) := Ch (E_i, F_i, G_i) +Σ1( E_i) +WK_i+ H_i + Maj(A_i , B_i, C_i) +Σ0( A_i);
    B_(i +1) := A_i;
    C_(i +1) := B_i ;
    D_(i +1) := C_i;
    E_(i +1) := Ch (E_i, F_i, G_i) +Σ1( E_i) +WK_i+ H_i + D_i;
    F_(i +1) := E_i ;
    G_(i +1) := F_i;
    H_(i +1) := G_i;
ENDFOR

DEST[127:96] := A_2;
DEST[95:64] := B_2;
DEST[63:32] := E_2;
DEST[31:0] := F_2;
```

Intel C/C++ Compiler Intrinsic Equivalent

SHA256RNDS2 __m128i _mm_sha256rnds2_epu32(__m128i, __m128i, __m128i);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions."

SHLD—Double Precision Shift Left

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F A4 /r ib	SHLD r/m16, r16, imm8	MRI	Valid	Valid	Shift r/m16 to left imm8 places while shifting bits from r16 in from the right.
0F A5 /r	SHLD r/m16, r16, CL	MRC	Valid	Valid	Shift r/m16 to left CL places while shifting bits from r16 in from the right.
0F A4 /r ib	SHLD r/m32, r32, imm8	MRI	Valid	Valid	Shift r/m32 to left imm8 places while shifting bits from r32 in from the right.
REX.W + 0F A4 /r ib	SHLD r/m64, r64, imm8	MRI	Valid	N.E.	Shift r/m64 to left imm8 places while shifting bits from r64 in from the right.
0F A5 /r	SHLD r/m32, r32, CL	MRC	Valid	Valid	Shift r/m32 to left CL places while shifting bits from r32 in from the right.
REX.W + 0F A5 /r	SHLD r/m64, r64, CL	MRC	Valid	N.E.	Shift r/m64 to left CL places while shifting bits from r64 in from the right.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MRI	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
MRC	ModRM:r/m (w)	ModRM:reg (r)	CL	N/A

Description

The SHLD instruction is used for multi-precision shifts of 64 bits or more.

The instruction shifts the first operand (destination operand) to the left the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the right (starting with bit 0 of the destination operand).

The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be stored in an immediate byte or in the CL register. If the count operand is CL, the shift count is the logical AND of CL and a count mask. In non-64-bit modes and default 64-bit mode; only bits 0 through 4 of the count are used. This masks the count to a value between 0 and 31. If a count is greater than the operand size, the result is undefined.

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. If the count operand is 0, flags are not affected.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits (upgrading the count mask to 6 bits). See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF (In 64-Bit Mode and REX.W = 1)
    THEN COUNT := COUNT MOD 64;
    ELSE COUNT := COUNT MOD 32;
FI
SIZE := OperandSize;
tempDEST := DEST;
IF COUNT > SIZE
    THEN (* Bad parameters *)
        tempDEST is undefined;
```

```
        CF, OF, SF, ZF, AF, PF are undefined;
    ELSE IF COUNT > 0 (* Perform the shift *)
        CF := BIT[tempDEST, SIZE – COUNT];
        (* Last bit shifted out on exit *)
        FOR i := SIZE – 1 DOWN TO COUNT
            DO
                Bit(tempDEST, i) := Bit(tempDEST, i – COUNT);
            OD;
        FOR i := COUNT – 1 DOWN TO 0
            DO
                BIT[tempDEST, i] := BIT[SRC, i – COUNT + SIZE];
            OD;
FI;
DEST := tempDEST;
```

Flags Affected

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand and the SF, ZF, and PF flags are set according to the value of the result. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. For shifts greater than 1 bit, the OF flag is undefined. If a shift occurs, the AF flag is undefined. If the count operand is 0, the flags are not affected. If the count is greater than the operand size, the flags are undefined.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.

#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

SHRD—Double Precision Shift Right

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F AC /r ib	SHRD r/m16, r16, imm8	MRI	Valid	Valid	Shift r/m16 to right imm8 places while shifting bits from r16 in from the left.
0F AD /r	SHRD r/m16, r16, CL	MRC	Valid	Valid	Shift r/m16 to right CL places while shifting bits from r16 in from the left.
0F AC /r ib	SHRD r/m32, r32, imm8	MRI	Valid	Valid	Shift r/m32 to right imm8 places while shifting bits from r32 in from the left.
REX.W + 0F AC /r ib	SHRD r/m64, r64, imm8	MRI	Valid	N.E.	Shift r/m64 to right imm8 places while shifting bits from r64 in from the left.
0F AD /r	SHRD r/m32, r32, CL	MRC	Valid	Valid	Shift r/m32 to right CL places while shifting bits from r32 in from the left.
REX.W + 0F AD /r	SHRD r/m64, r64, CL	MRC	Valid	N.E.	Shift r/m64 to right CL places while shifting bits from r64 in from the left.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MRI	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
MRC	ModRM:r/m (w)	ModRM:reg (r)	CL	N/A

Description

The SHRD instruction is useful for multi-precision shifts of 64 bits or more.

The instruction shifts the first operand (destination operand) to the right the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the left (starting with the most significant bit of the destination operand).

The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be stored in an immediate byte or the CL register. If the count operand is CL, the shift count is the logical AND of CL and a count mask. In non-64-bit modes and default 64-bit mode, the width of the count mask is 5 bits. Only bits 0 through 4 of the count register are used (masking the count to a value between 0 and 31). If the count is greater than the operand size, the result is undefined.

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. If the count operand is 0, flags are not affected.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits (upgrading the count mask to 6 bits). See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF (In 64-Bit Mode and REX.W = 1)
    THEN COUNT := COUNT MOD 64;
    ELSE COUNT := COUNT MOD 32;
FI
SIZE := OperandSize;
tempDEST := DEST;
IF COUNT > SIZE
    THEN (* Bad parameters *)
        tempDEST is undefined;
```

```
        CF, OF, SF, ZF, AF, PF are undefined;
    ELSE IF COUNT >0 (* Perform the shift *)
        CF := BIT[tempDEST, COUNT – 1]; (* Last bit shifted out on exit *)
        FOR i := 0 TO SIZE – 1 – COUNT
            DO
                BIT[tempDEST, i] := BIT[tempDEST, i + COUNT];
            OD;
        FOR i := SIZE – COUNT TO SIZE – 1
            DO
                BIT[tempDEST, i] := BIT[SRC, i + COUNT – SIZE];
            OD;
FI;
DEST := tempDEST;
```

Flags Affected

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand and the SF, ZF, and PF flags are set according to the value of the result. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. For shifts greater than 1 bit, the OF flag is undefined. If a shift occurs, the AF flag is undefined. If the count operand is 0, the flags are not affected. If the count is greater than the operand size, the flags are undefined.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.

#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

SHUFPD—Packed Interleave Shuffle of Pairs of Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F C6 /r ib SHUFPD xmm1, xmm2/m128, imm8	A	V/V	SSE2	Shuffle two pairs of double precision floating-point values from xmm1 and xmm2/m128 using imm8 to select from each pair, interleaved result is stored in xmm1.
VEX.128.66.0F.WIG C6 /r ib VSHUFPD xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Shuffle two pairs of double precision floating-point values from xmm2 and xmm3/m128 using imm8 to select from each pair, interleaved result is stored in xmm1.
VEX.256.66.0F.WIG C6 /r ib VSHUFPD ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Shuffle four pairs of double precision floating-point values from ymm2 and ymm3/m256 using imm8 to select from each pair, interleaved result is stored in xmm1.
EVEX.128.66.0F.W1 C6 /r ib VSHUFPD xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shuffle two pairs of double precision floating-point values from xmm2 and xmm3/m128/m64bcst using imm8 to select from each pair. store interleaved results in xmm1 subject to writemask k1.
EVEX.256.66.0F.W1 C6 /r ib VSHUFPD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shuffle four pairs of double precision floating-point values from ymm2 and ymm3/m256/m64bcst using imm8 to select from each pair. store interleaved results in ymm1 subject to writemask k1.
EVEX.512.66.0F.W1 C6 /r ib VSHUFPD zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	C	V/V	AVX512F OR AVX10.1	Shuffle eight pairs of double precision floating-point values from zmm2 and zmm3/m512/m64bcst using imm8 to select from each pair. store interleaved results in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Selects a double precision floating-point value of an input pair using a bit control and move to a designated element of the destination operand. The low-to-high order of double precision element of the destination operand is interleaved between the first source operand and the second source operand at the granularity of input pair of 128 bits. Each bit in the imm8 byte, starting from bit 0, is the select control of the corresponding element of the destination to received the shuffled result of an input pair.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location The destination operand is a ZMM/YMM/XMM register updated according to the writemask. The select controls are the lower 8/4/2 bits of the imm8 byte.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The select controls are the bit 3:0 of the imm8 byte, imm8[7:4] are ignored.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of

the corresponding ZMM register destination are zeroed. The select controls are the bit 1:0 of the imm8 byte, imm8[7:2) are ignored.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination operand and the first source operand is the same and is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. The select controls are the bit 1:0 of the imm8 byte, imm8[7:2) are ignored.

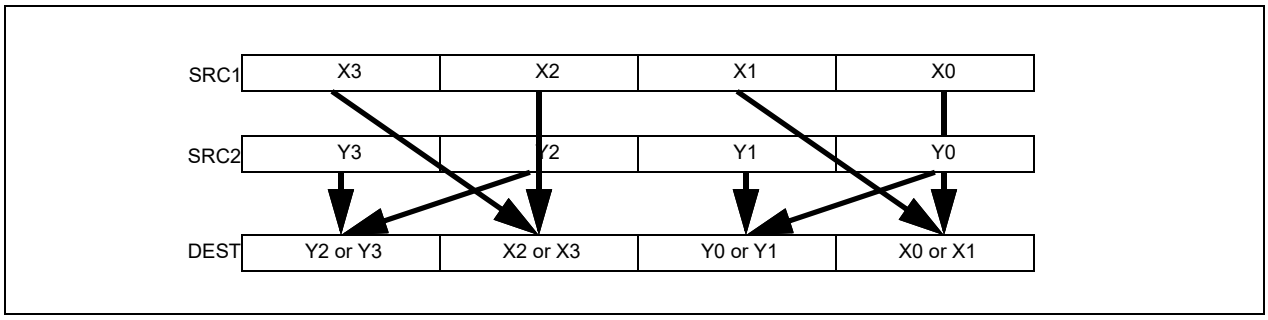


Figure 4-25. 256-bit VSHUFPD Operation of Four Pairs of Double Precision Floating-Point Values

Operation

VSHUFPD (EVEX Encoded Versions When SRC2 is a Vector Register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF IMM0[0] = 0
    THEN TMP_DEST[63:0] := SRC1[63:0]
    ELSE TMP_DEST[63:0] := SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN TMP_DEST[127:64] := SRC2[63:0]
    ELSE TMP_DEST[127:64] := SRC2[127:64] FI;
IF VL >= 256
    IF IMM0[2] = 0
        THEN TMP_DEST[191:128] := SRC1[191:128]
        ELSE TMP_DEST[191:128] := SRC1[255:192] FI;
    IF IMM0[3] = 0
        THEN TMP_DEST[255:192] := SRC2[191:128]
        ELSE TMP_DEST[255:192] := SRC2[255:192] FI;
FI;
IF VL >= 512
    IF IMM0[4] = 0
        THEN TMP_DEST[319:256] := SRC1[319:256]
        ELSE TMP_DEST[319:256] := SRC1[383:320] FI;
    IF IMM0[5] = 0
        THEN TMP_DEST[383:320] := SRC2[319:256]
        ELSE TMP_DEST[383:320] := SRC2[383:320] FI;
    IF IMM0[6] = 0
        THEN TMP_DEST[447:384] := SRC1[447:384]
        ELSE TMP_DEST[447:384] := SRC1[511:448] FI;
    IF IMM0[7] = 0
        THEN TMP_DEST[511:448] := SRC2[447:384]
        ELSE TMP_DEST[511:448] := SRC2[511:448] FI;
FI;
FOR j := 0 TO KL-1
    i := j * 64
```

```
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VSHUFPD (EVEX Encoded Versions When SRC2 is Memory)

(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i]
    FI;
ENDFOR;
IF IMM0[0] = 0
    THEN TMP_DEST[63:0] := SRC1[63:0]
    ELSE TMP_DEST[63:0] := SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN TMP_DEST[127:64] := TMP_SRC2[63:0]
    ELSE TMP_DEST[127:64] := TMP_SRC2[127:64] FI;
IF VL >= 256
    IF IMM0[2] = 0
        THEN TMP_DEST[191:128] := SRC1[191:128]
        ELSE TMP_DEST[191:128] := SRC1[255:192] FI;
    IF IMM0[3] = 0
        THEN TMP_DEST[255:192] := TMP_SRC2[191:128]
        ELSE TMP_DEST[255:192] := TMP_SRC2[255:192] FI;
FI;
IF VL >= 512
    IF IMM0[4] = 0
        THEN TMP_DEST[319:256] := SRC1[319:256]
        ELSE TMP_DEST[319:256] := SRC1[383:320] FI;
    IF IMM0[5] = 0
        THEN TMP_DEST[383:320] := TMP_SRC2[319:256]
        ELSE TMP_DEST[383:320] := TMP_SRC2[383:320] FI;
    IF IMM0[6] = 0
        THEN TMP_DEST[447:384] := SRC1[447:384]
        ELSE TMP_DEST[447:384] := SRC1[511:448] FI;
    IF IMM0[7] = 0
        THEN TMP_DEST[511:448] := TMP_SRC2[447:384]
        ELSE TMP_DEST[511:448] := TMP_SRC2[511:448] FI;
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
```

```
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VSHUFPD (VEX.256 Encoded Version)

```
IF IMM0[0] = 0
    THEN DEST[63:0] := SRC1[63:0]
    ELSE DEST[63:0] := SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN DEST[127:64] := SRC2[63:0]
    ELSE DEST[127:64] := SRC2[127:64] FI;
IF IMM0[2] = 0
    THEN DEST[191:128] := SRC1[191:128]
    ELSE DEST[191:128] := SRC1[255:192] FI;
IF IMM0[3] = 0
    THEN DEST[255:192] := SRC2[191:128]
    ELSE DEST[255:192] := SRC2[255:192] FI;
DEST[MAXVL-1:256] (Unmodified)
```

VSHUFPD (VEX.128 Encoded Version)

```
IF IMM0[0] = 0
    THEN DEST[63:0] := SRC1[63:0]
    ELSE DEST[63:0] := SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN DEST[127:64] := SRC2[63:0]
    ELSE DEST[127:64] := SRC2[127:64] FI;
DEST[MAXVL-1:128] := 0
```

VSHUFPD (128-bit Legacy SSE Version)

```
IF IMM0[0] = 0
    THEN DEST[63:0] := SRC1[63:0]
    ELSE DEST[63:0] := SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN DEST[127:64] := SRC2[63:0]
    ELSE DEST[127:64] := SRC2[127:64] FI;
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VSHUFPD __m512d _mm512_shuffle_pd(__m512d a, __m512d b, int imm);
VSHUFPD __m512d _mm512_mask_shuffle_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int imm);
VSHUFPD __m512d _mm512_maskz_shuffle_pd(__mmask8 k, __m512d a, __m512d b, int imm);
VSHUFPD __m256d _mm256_shuffle_pd (__m256d a, __m256d b, const int select);
VSHUFPD __m256d _mm256_mask_shuffle_pd(__m256d s, __mmask8 k, __m256d a, __m256d b, int imm);
VSHUFPD __m256d _mm256_maskz_shuffle_pd(__mmask8 k, __m256d a, __m256d b, int imm);
SHUFPD __m128d _mm_shuffle_pd (__m128d a, __m128d b, const int select);
VSHUFPD __m128d _mm_mask_shuffle_pd(__m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VSHUFPD __m128d _mm_maskz_shuffle_pd(__mmask8 k, __m128d a, __m128d b, int imm);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-52, "Type E4NF Class Exception Conditions."

SHUFPS—Packed Interleave Shuffle of Quadruplets of Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C6 /r ib SHUFPS xmm1, xmm3/m128, imm8	A	V/V	SSE	Select from quadruplet of single precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1.
VEX.128.0F.WIG C6 /r ib VSHUFPS xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Select from quadruplet of single precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1.
VEX.256.0F.WIG C6 /r ib VSHUFPS ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Select from quadruplet of single precision floating-point values in ymm2 and ymm3/m256 using imm8, interleaved result pairs are stored in ymm1.
EVEX.128.0F.W0 C6 /r ib VSHUFPS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Select from quadruplet of single precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1, subject to writemask k1.
EVEX.256.0F.W0 C6 /r ib VSHUFPS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Select from quadruplet of single precision floating-point values in ymm2 and ymm3/m256 using imm8, interleaved result pairs are stored in ymm1, subject to writemask k1.
EVEX.512.0F.W0 C6 /r ib VSHUFPS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	C	V/V	AVX512F OR AVX10.1	Select from quadruplet of single precision floating-point values in zmm2 and zmm3/m512 using imm8, interleaved result pairs are stored in zmm1, subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Selects a single precision floating-point value of an input quadruplet using a two-bit control and move to a designated element of the destination operand. Each 64-bit element-pair of a 128-bit lane of the destination operand is interleaved between the corresponding lane of the first source operand and the second source operand at the granularity 128 bits. Each two bits in the imm8 byte, starting from bit 0, is the select control of the corresponding element of a 128-bit lane of the destination to received the shuffled result of an input quadruplet. The two lower elements of a 128-bit lane in the destination receives shuffle results from the quadruple of the first source operand. The next two elements of the destination receives shuffle results from the quadruple of the second source operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask. imm8[7:0] provides 4 select controls for each applicable 128-bit lane of the destination.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Imm8[7:0] provides 4 select controls for the high and low 128-bit of the destination.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed. Imm8[7:0] provides 4 select controls for each element of the destination.

128-bit Legacy SSE version: The source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. Imm8[7:0] provides 4 select controls for each element of the destination.

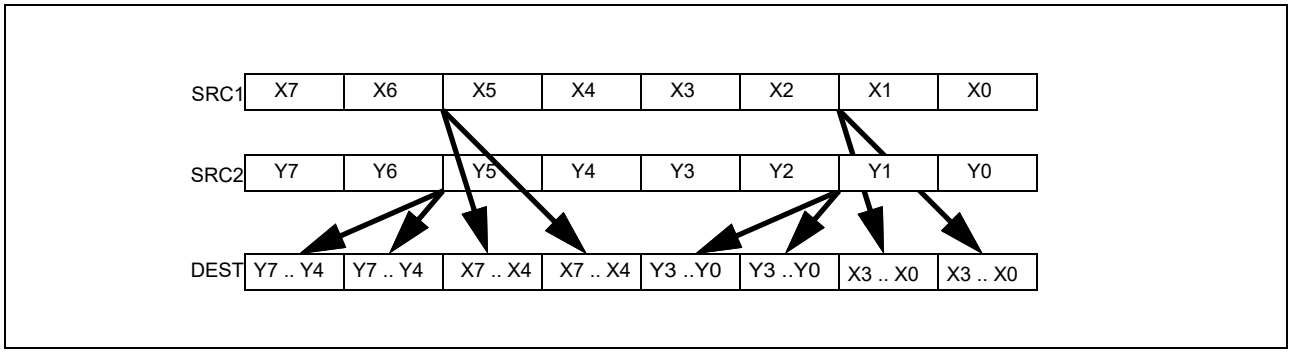


Figure 4-26. 256-bit VSHUFPS Operation of Selection from Input Quadruplet and Pair-wise Interleaved Result

Operation

```
Select4(SRC, control) {
CASE (control[1:0]) OF
    0:    TMP := SRC[31:0];
    1:    TMP := SRC[63:32];
    2:    TMP := SRC[95:64];
    3:    TMP := SRC[127:96];
ESAC;
RETURN TMP
}
```

VPSHUFPS (EVEX Encoded Versions When SRC2 is a Vector Register)
(KL, VL) = (4, 128), (8, 256), (16, 512)

```
TMP_DEST[31:0] := Select4(SRC1[127:0], imm8[1:0]);
TMP_DEST[63:32] := Select4(SRC1[127:0], imm8[3:2]);
TMP_DEST[95:64] := Select4(SRC2[127:0], imm8[5:4]);
TMP_DEST[127:96] := Select4(SRC2[127:0], imm8[7:6]);
IF VL >= 256
    TMP_DEST[159:128] := Select4(SRC1[255:128], imm8[1:0]);
    TMP_DEST[191:160] := Select4(SRC1[255:128], imm8[3:2]);
    TMP_DEST[223:192] := Select4(SRC2[255:128], imm8[5:4]);
    TMP_DEST[255:224] := Select4(SRC2[255:128], imm8[7:6]);
FI;
IF VL >= 512
    TMP_DEST[287:256] := Select4(SRC1[383:256], imm8[1:0]);
    TMP_DEST[319:288] := Select4(SRC1[383:256], imm8[3:2]);
    TMP_DEST[351:320] := Select4(SRC2[383:256], imm8[5:4]);
    TMP_DEST[383:352] := Select4(SRC2[383:256], imm8[7:6]);
    TMP_DEST[415:384] := Select4(SRC1[511:384], imm8[1:0]);
    TMP_DEST[447:416] := Select4(SRC1[511:384], imm8[3:2]);
    TMP_DEST[479:448] := Select4(SRC2[511:384], imm8[5:4]);
    TMP_DEST[511:480] := Select4(SRC2[511:384], imm8[7:6]);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
```

```
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSHUFPS (EVEX Encoded Versions When SRC2 is Memory)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+31:i] := SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i]
    FI;
ENDFOR;
TMP_DEST[31:0] := Select4(SRC1[127:0], imm8[1:0]);
TMP_DEST[63:32] := Select4(SRC1[127:0], imm8[3:2]);
TMP_DEST[95:64] := Select4(TMP_SRC2[127:0], imm8[5:4]);
TMP_DEST[127:96] := Select4(TMP_SRC2[127:0], imm8[7:6]);
IF VL >= 256
    TMP_DEST[159:128] := Select4(SRC1[255:128], imm8[1:0]);
    TMP_DEST[191:160] := Select4(SRC1[255:128], imm8[3:2]);
    TMP_DEST[223:192] := Select4(TMP_SRC2[255:128], imm8[5:4]);
    TMP_DEST[255:224] := Select4(TMP_SRC2[255:128], imm8[7:6]);
FI;
IF VL >= 512
    TMP_DEST[287:256] := Select4(SRC1[383:256], imm8[1:0]);
    TMP_DEST[319:288] := Select4(SRC1[383:256], imm8[3:2]);
    TMP_DEST[351:320] := Select4(TMP_SRC2[383:256], imm8[5:4]);
    TMP_DEST[383:352] := Select4(TMP_SRC2[383:256], imm8[7:6]);
    TMP_DEST[415:384] := Select4(SRC1[511:384], imm8[1:0]);
    TMP_DEST[447:416] := Select4(SRC1[511:384], imm8[3:2]);
    TMP_DEST[479:448] := Select4(TMP_SRC2[511:384], imm8[5:4]);
    TMP_DEST[511:480] := Select4(TMP_SRC2[511:384], imm8[7:6]);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VSHUFPS (VEX.256 Encoded Version)

```
DEST[31:0] := Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] := Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] := Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] := Select4(SRC2[127:0], imm8[7:6]);
DEST[159:128] := Select4(SRC1[255:128], imm8[1:0]);
DEST[191:160] := Select4(SRC1[255:128], imm8[3:2]);
DEST[223:192] := Select4(SRC2[255:128], imm8[5:4]);
DEST[255:224] := Select4(SRC2[255:128], imm8[7:6]);
DEST[MAXVL-1:256] := 0
```

VSHUFPS (VEX.128 Encoded Version)

```
DEST[31:0] := Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] := Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] := Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] := Select4(SRC2[127:0], imm8[7:6]);
DEST[MAXVL-1:128] := 0
```

SHUFPS (128-bit Legacy SSE Version)

```
DEST[31:0] := Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] := Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] := Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] := Select4(SRC2[127:0], imm8[7:6]);
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VSHUFPS __m512 _mm512_shuffle_ps(__m512 a, __m512 b, int imm);
VSHUFPS __m512 _mm512_mask_shuffle_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VSHUFPS __m512 _mm512_maskz_shuffle_ps(__mmask16 k, __m512 a, __m512 b, int imm);
VSHUFPS __m256 _mm256_shuffle_ps (__m256 a, __m256 b, const int select);
VSHUFPS __m256 _mm256_mask_shuffle_ps(__m256 s, __mmask8 k, __m256 a, __m256 b, int imm);
VSHUFPS __m256 _mm256_maskz_shuffle_ps(__mmask8 k, __m256 a, __m256 b, int imm);
SHUFPS __m128 _mm_shuffle_ps (__m128 a, __m128 b, const int select);
VSHUFPS __m128 _mm_mask_shuffle_ps(__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VSHUFPS __m128 _mm_maskz_shuffle_ps(__mmask8 k, __m128 a, __m128 b, int imm);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-52, “Type E4NF Class Exception Conditions.”

SIDT—Store Interrupt Descriptor Table Register

Opcode[1]	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 /1	SIDT *m*	M	Valid	Valid	Store IDTR to *m*.

NOTES:

1. See the IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	N/A	N/A	N/A

Description

Stores the content the interrupt descriptor table register (IDTR) in the destination operand. The destination operand specifies a 6-byte memory location.

In non-64-bit modes, the 16-bit limit field of the register is stored in the low 2 bytes of the memory location and the 32-bit base address is stored in the high 4 bytes.

In 64-bit mode, the operand size fixed at 8+2 bytes. The instruction stores 8-byte base and 2-byte limit values.

SIDT is only useful in operating-system software; however, it can be used in application programs without causing an exception to be generated if CR4.UMIP = 0. See "LGDT/LIDT—Load Global/Interrupt Descriptor Table Register" in Chapter 3, Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, for information on loading the GDTR and IDTR.

IA-32 Architecture Compatibility

The 16-bit form of SIDT is compatible with the Intel 286 processor if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; processor generations later than the Intel 286 processor fill these bits with 0s.

Operation

```
IF instruction is SIDT
    THEN
        IF OperandSize =16 or OperandSize = 32 (* Legacy or Compatibility Mode *)
            THEN
                DEST[0:15] := IDTR(Limit);
                DEST[16:47] := IDTR(Base); FI; (* Full 32-bit base address stored *)
            ELSE (* 64-bit Mode *)
                DEST[0:15] := IDTR(Limit);
                DEST[16:79] := IDTR(Base); (* Full 64-bit base address stored *)
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If CR4.UMIP = 1 and CPL > 0.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If CR4.UMIP = 1.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#UD	If the LOCK prefix is used.
#GP(0)	If the memory address is in a non-canonical form.
	If CR4.UMIP = 1 and CPL > 0.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.

SLDT—Store Local Descriptor Table Register

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 00 /0	SLDT *r/m16*	M	Valid	Valid	Stores segment selector from LDTR in *r/m16*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	N/A	N/A	N/A

Description

Stores the segment selector from the local descriptor table register (LDTR) in the destination operand. The destination operand can be a general-purpose register or a memory location. The segment selector stored with this instruction points to the segment descriptor (located in the GDT) for the current LDT. This instruction can only be executed in protected mode.

Outside IA-32e mode, when the destination operand is a 32-bit register, the 16-bit segment selector is copied into the low-order 16 bits of the register. The high-order 16 bits of the register are cleared for the Pentium 4, Intel Xeon, and P6 family processors. They are undefined for Pentium, Intel486, and Intel386 processors. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of the operand size.

In compatibility mode, when the destination operand is a 32-bit register, the 16-bit segment selector is copied into the low-order 16 bits of the register. The high-order 16 bits of the register are cleared. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of the operand size.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). The behavior of SLDT with a 64-bit register is to zero-extend the 16-bit selector and store it in the register. If the destination is memory and operand size is 64, SLDT will write the 16-bit selector to memory as a 16-bit quantity, regardless of the operand size.

Operation

```
DEST := LDTR(SegmentSelector);
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If CR4.UMIP = 1 and CPL > 0.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	The SLDT instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The SLDT instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If CR4.UMIP = 1 and CPL > 0.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
#UD	If the LOCK prefix is used.

SMSW—Store Machine Status Word

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 /4	SMSW r/m16	M	Valid	Valid	Store machine status word to r/m16.
0F 01 /4	SMSW r32/m16	M	Valid	Valid	Store machine status word in low-order 16 bits of r32/m16; high-order 16 bits of r32 are undefined.
REX.W + 0F 01 /4	SMSW r64/m16	M	Valid	Valid	Store machine status word in low-order 16 bits of r64/m16; high-order 16 bits of r32 are undefined.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	N/A	N/A	N/A

Description

Stores the machine status word (bits 0 through 15 of control register CR0) into the destination operand. The destination operand can be a general-purpose register or a memory location.

In non-64-bit modes, when the destination operand is a 32-bit register, the low-order 16 bits of register CR0 are copied into the low-order 16 bits of the register and the high-order 16 bits are undefined. When the destination operand is a memory location, the low-order 16 bits of register CR0 are written to memory as a 16-bit quantity, regardless of the operand size.

In 64-bit mode, the behavior of the SMSW instruction is defined by the following examples:

- SMSW r16 operand size 16, store CR0[15:0] in r16
- SMSW r32 operand size 32, zero-extend CR0[31:0], and store in r32
- SMSW r64 operand size 64, zero-extend CR0[63:0], and store in r64
- SMSW m16 operand size 16, store CR0[15:0] in m16
- SMSW m16 operand size 32, store CR0[15:0] in m16 (not m32)
- SMSW m16 operands size 64, store CR0[15:0] in m16 (not m64)

SMSW is only useful in operating-system software. However, it is not a privileged instruction and can be used in application programs if CR4.UMIP = 0. It is provided for compatibility with the Intel 286 processor. Programs and procedures intended to run on IA-32 and Intel 64 processors beginning with the Intel386 processors should use the MOV CR instruction to load the machine status word.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 27 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C, for more information about the behavior of this instruction in VMX non-root operation.

Operation

```
DEST := CR0[15:0];
(* Machine status word *)
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If CR4.UMIP = 1 and CPL > 0.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If CR4.UMIP = 1.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
	If CR4.UMIP = 1 and CPL > 0.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
#UD	If the LOCK prefix is used.

SQRTPD—Square Root of Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 51 /r SQRTPD xmm1, xmm2/m128	A	V/V	SSE2	Computes Square Roots of the packed double precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.128.66.0F.WIG 51 /r VSQRTPD xmm1, xmm2/m128	A	V/V	AVX	Computes Square Roots of the packed double precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.256.66.0F.WIG 51 /r VSQRTPD ymm1, ymm2/m256	A	V/V	AVX	Computes Square Roots of the packed double precision floating-point values in ymm2/m256 and stores the result in ymm1.
EVEX.128.66.0F.W1 51 /r VSQRTPD xmm1 {k1}{z}, xmm2/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Computes Square Roots of the packed double precision floating-point values in xmm2/m128/m64bcst and stores the result in xmm1 subject to writemask k1.
EVEX.256.66.0F.W1 51 /r VSQRTPD ymm1 {k1}{z}, ymm2/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Computes Square Roots of the packed double precision floating-point values in ymm2/m256/m64bcst and stores the result in ymm1 subject to writemask k1.
EVEX.512.66.0F.W1 51 /r VSQRTPD zmm1 {k1}{z}, zmm2/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Computes Square Roots of the packed double precision floating-point values in zmm2/m512/m64bcst and stores the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Performs a SIMD computation of the square roots of the two, four or eight packed double precision floating-point values in the source operand (the second operand) stores the packed double precision floating-point results in the destination operand (the first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

VSQRTPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC *is register*)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+63:i] := SQRT(SRC[63:0])
                ELSE DEST[i+63:i] := SQRT(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VSQRTPD (VEX.256 Encoded Version)

```
DEST[63:0] := SQRT(SRC[63:0])
DEST[127:64] := SQRT(SRC[127:64])
DEST[191:128] := SQRT(SRC[191:128])
DEST[255:192] := SQRT(SRC[255:192])
DEST[MAXVL-1:256] := 0
```

.

VSQRTPD (VEX.128 Encoded Version)

```
DEST[63:0] := SQRT(SRC[63:0])
DEST[127:64] := SQRT(SRC[127:64])
DEST[MAXVL-1:128] := 0
```

SQRTPD (128-bit Legacy SSE Version)

```
DEST[63:0] := SQRT(SRC[63:0])
DEST[127:64] := SQRT(SRC[127:64])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VSQRTPD __m512d _mm512_sqrt_round_pd(__m512d a, int r);
VSQRTPD __m512d _mm512_mask_sqrt_round_pd(__m512d s, __mmask8 k, __m512d a, int r);
VSQRTPD __m512d _mm512_maskz_sqrt_round_pd(__mmask8 k, __m512d a, int r);
VSQRTPD __m256d _mm256_sqrt_pd (__m256d a);
VSQRTPD __m256d _mm256_mask_sqrt_pd(__m256d s, __mmask8 k, __m256d a, int r);
VSQRTPD __m256d _mm256_maskz_sqrt_pd(__mmask8 k, __m256d a, int r);
SQRTPD __m128d _mm_sqrt_pd (__m128d a);
VSQRTPD __m128d _mm_mask_sqrt_pd(__m128d s, __mmask8 k, __m128d a, int r);
VSQRTPD __m128d _mm_maskz_sqrt_pd(__mmask8 k, __m128d a, int r);

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, "Type 2 Class Exception Conditions," additionally:

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Table 2-48, "Type E2 Class Exception Conditions," additionally:

#UD	If EVEX.vvvv != 1111B.

SQRTPS—Square Root of Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 51 /r SQRTPS xmm1, xmm2/m128	A	V/V	SSE	Computes Square Roots of the packed single precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.128.0F.WIG 51 /r VSQRTPS xmm1, xmm2/m128	A	V/V	AVX	Computes Square Roots of the packed single precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.256.0F.WIG 51/r VSQRTPS ymm1, ymm2/m256	A	V/V	AVX	Computes Square Roots of the packed single precision floating-point values in ymm2/m256 and stores the result in ymm1.
EVEX.128.0F.W0 51 /r VSQRTPS xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Computes Square Roots of the packed single precision floating-point values in xmm2/m128/m32bcst and stores the result in xmm1 subject to writemask k1.
EVEX.256.0F.W0 51 /r VSQRTPS ymm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Computes Square Roots of the packed single precision floating-point values in ymm2/m256/m32bcst and stores the result in ymm1 subject to writemask k1.
EVEX.512.0F.W0 51/r VSQRTPS zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Computes Square Roots of the packed single precision floating-point values in zmm2/m512/m32bcst and stores the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Performs a SIMD computation of the square roots of the four, eight or sixteen packed single precision floating-point values in the source operand (second operand) stores the packed single precision floating-point results in the destination operand.

EVEX.512 encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

VSQRTPS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC *is register*)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+31:i] := SQRT(SRC[31:0])
                ELSE DEST[i+31:i] := SQRT(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VSQRTPS (VEX.256 Encoded Version)

```
DEST[31:0] := SQRT(SRC[31:0])
DEST[63:32] := SQRT(SRC[63:32])
DEST[95:64] := SQRT(SRC[95:64])
DEST[127:96] := SQRT(SRC[127:96])
DEST[159:128] := SQRT(SRC[159:128])
DEST[191:160] := SQRT(SRC[191:160])
DEST[223:192] := SQRT(SRC[223:192])
DEST[255:224] := SQRT(SRC[255:224])
```

VSQRTPS (VEX.128 Encoded Version)

```
DEST[31:0] := SQRT(SRC[31:0])
DEST[63:32] := SQRT(SRC[63:32])
DEST[95:64] := SQRT(SRC[95:64])
DEST[127:96] := SQRT(SRC[127:96])
DEST[MAXVL-1:128] := 0
```

SQRTPS (128-bit Legacy SSE Version)

```
DEST[31:0] := SQRT(SRC[31:0])
DEST[63:32] := SQRT(SRC[63:32])
DEST[95:64] := SQRT(SRC[95:64])
DEST[127:96] := SQRT(SRC[127:96])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VSQRTPS __m512 _mm512_sqrt_round_ps(__m512 a, int r);
VSQRTPS __m512 _mm512_mask_sqrt_round_ps(__m512 s, __mmask16 k, __m512 a, int r);
VSQRTPS __m512 _mm512_maskz_sqrt_round_ps(__mmask16 k, __m512 a, int r);
VSQRTPS __m256 _mm256_sqrt_ps (__m256 a);
VSQRTPS __m256 _mm256_mask_sqrt_ps(__m256 s, __mmask8 k, __m256 a, int r);
VSQRTPS __m256 _mm256_maskz_sqrt_ps(__mmask8 k, __m256 a, int r);
SQRTPS __m128 _mm_sqrt_ps (__m128 a);
VSQRTPS __m128 _mm_mask_sqrt_ps(__m128 s, __mmask8 k, __m128 a, int r);
VSQRTPS __m128 _mm_maskz_sqrt_ps(__mmask8 k, __m128 a, int r);

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, "Type 2 Class Exception Conditions," additionally:

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Table 2-48, "Type E2 Class Exception Conditions," additionally:

#UD	If EVEX.vvvv != 1111B.

SQRTSD—Compute Square Root of Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 51/r SQRTSD xmm1,xmm2/m64	A	V/V	SSE2	Computes square root of the low double precision floating-point value in xmm2/m64 and stores the results in xmm1.
VEX.LIG.F2.0F.WIG 51/r VSQRTSD xmm1,xmm2, xmm3/m64	B	V/V	AVX	Computes square root of the low double precision floating-point value in xmm3/m64 and stores the results in xmm1. Also, upper double precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].
EVEX.LLIG.F2.0F.W1 51/r VSQRTSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F OR AVX10.1	Computes square root of the low double precision floating-point value in xmm3/m64 and stores the results in xmm1 under writemask k1. Also, upper double precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Computes the square root of the low double precision floating-point value in the second source operand and stores the double precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. The quadword at bits 127:64 of the destination operand remains unchanged. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits 127:64 of the destination operand are copied from the corresponding bits of the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the write-mask.

Software should ensure VSQRTSD is encoded with VEX.L=0. Encoding VSQRTSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VSQRTSD (EVEX Encoded Version)

```
IF (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := SQRT(SRC2[63:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VSQRTSD (VEX.128 Encoded Version)

```
DEST[63:0] := SQRT(SRC2[63:0])
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

SQRTSD (128-bit Legacy SSE Version)

```
DEST[63:0] := SQRT(SRC[63:0])
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VSQRTSD __m128d _mm_sqrt_round_sd(__m128d a, __m128d b, int r);
VSQRTSD __m128d _mm_mask_sqrt_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int r);
VSQRTSD __m128d _mm_maskz_sqrt_round_sd(__mmask8 k, __m128d a, __m128d b, int r);
SQRTSD __m128d _mm_sqrt_sd (__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-49, “Type E3 Class Exception Conditions.”

SQRTSS—Compute Square Root of Scalar Single Precision Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 51 /r SQRTSS xmm1, xmm2/m32	A	V/V	SSE	Computes square root of the low single precision floating-point value in xmm2/m32 and stores the results in xmm1.
VEX.LIG.F3.0F.WIG 51 /r VSQRTSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Computes square root of the low single precision floating-point value in xmm3/m32 and stores the results in xmm1. Also, upper single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].
EVEX.LLIG.F3.0F.W0 51 /r VSQRTSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	C	V/V	AVX512F OR AVX10.1	Computes square root of the low single precision floating-point value in xmm3/m32 and stores the results in xmm1 under writemask k1. Also, upper single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Computes the square root of the low single precision floating-point value in the second source operand and stores the single precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands is an XMM register.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits 127:32 of the destination operand are copied from the corresponding bits of the first source operand. Bits (MAXVL-1:128) of the destination ZMM register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the write-mask.

Software should ensure VSQRTSS is encoded with VEX.L=0. Encoding VSQRTSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VSQRTSS (EVEX Encoded Version)

```
IF (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := SQRT(SRC2[31:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VSQRTSS (VEX.128 Encoded Version)

```
DEST[31:0] := SQRT(SRC2[31:0])
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

SQRTSS (128-bit Legacy SSE Version)

```
DEST[31:0] := SQRT(SRC2[31:0])
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VSQRTSS __m128 _mm_sqrt_round_ss(__m128 a, __m128 b, int r);
VSQRTSS __m128 _mm_mask_sqrt_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int r);
VSQRTSS __m128 _mm_maskz_sqrt_round_ss(__mmask8 k, __m128 a, __m128 b, int r);
SQRTSS __m128 _mm_sqrt_ss(__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-49, “Type E3 Class Exception Conditions.”

STAC—Set AC Flag in EFLAGS Register

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 CB STAC	ZO	V/V	SMAP	Set the AC flag in the EFLAGS register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Sets the AC flag bit in EFLAGS register. This may enable alignment checking of user-mode data accesses. This allows explicit supervisor-mode data accesses to user-mode pages even if the SMAP bit is set in the CR4 register.

This instruction's operation is the same in non-64-bit modes and 64-bit mode. Attempts to execute STAC when CPL > 0 cause #UD.

Operation

```
EFLAGS.AC := 1;
```

Flags Affected

AC set. Other flags are unaffected.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If the CPL > 0.
	If CPUID.07H.00H:EBX.SMAP[20] = 0.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.07H.00H:EBX.SMAP[20] = 0.

Virtual-8086 Mode Exceptions

#UD	The STAC instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.
	If the CPL > 0.
	If CPUID.07H.00H:EBX.SMAP[20] = 0.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If the CPL > 0.
	If CPUID.07H.00H:EBX.SMAP[20] = 0.

STC—Set Carry Flag

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F9	STC	ZO	Valid	Valid	Set CF flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Sets the CF flag in the EFLAGS register. Operation is the same in all modes.

Operation

```
CF := 1;
```

Flags Affected

The CF flag is set. The OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

STD—Set Direction Flag

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
FD	STD	ZO	Valid	Valid	Set DF flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Sets the DF flag in the EFLAGS register. When the DF flag is set to 1, string operations decrement the index registers (ESI and/or EDI). Operation is the same in all modes.

Operation

```
DF := 1;
```

Flags Affected

The DF flag is set. The CF, OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

STI—Set Interrupt Flag

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FB	STI	ZO	Valid	Valid	Set interrupt flag; external, maskable interrupts enabled at the end of the next instruction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

In most cases, STI sets the interrupt flag (IF) in the EFLAGS register. This allows the processor to respond to maskable hardware interrupts.

If IF = 0, maskable hardware interrupts remain inhibited on the instruction boundary following an execution of STI. (The delayed effect of this instruction is provided to allow interrupts to be enabled just before returning from a procedure or subroutine. For instance, if an STI instruction is followed by an RET instruction, the RET instruction is allowed to execute before external interrupts are recognized. No interrupts can be recognized if an execution of CLI immediately follow such an execution of STI.) The inhibition ends after delivery of another event (e.g., exception) or the execution of the next instruction.

The IF flag and the STI and CLI instructions do not prohibit the generation of exceptions and nonmaskable interrupts (NMIs). However, NMIs (and system-management interrupts) may be inhibited on the instruction boundary following an execution of STI that begins with IF = 0.

Operation is different in two modes defined as follows:

- **PVI mode** (protected-mode virtual interrupts): CR0.PE = 1, EFLAGS.VM = 0, CPL = 3, and CR4.PVI = 1;
- **VME mode** (virtual-8086 mode extensions): CR0.PE = 1, EFLAGS.VM = 1, and CR4.VME = 1.

If IOPL < 3, EFLAGS.VIP = 1, and either VME mode or PVI mode is active, STI sets the VIF flag in the EFLAGS register, leaving IF unaffected.

Table 4-22 indicates the action of the STI instruction depending on the processor operating mode, IOPL, CPL, and EFLAGS.VIP.

Table 4-22. Decision Table for STI Results

Mode	IOPL	EFLAGS.VIP	STI Result
Real-address	X[1]	X	IF = 1
Protected, not PVI[2]	≥ CPL	X	IF = 1
	< CPL	X	#GP fault
Protected, PVI[3]	3	X	IF = 1
	0–2	0	VIF = 1
		1	#GP fault
Virtual-8086, not VME[3]	3	X	IF = 1
	0–2	X	#GP fault
Virtual-8086, VME[3]	3	X	IF = 1
	0–2	0	VIF = 1
		1	#GP fault

NOTES:

1. X = This setting has no effect on instruction operation.

2. For this table, “protected mode” applies whenever CR0.PE = 1 and EFLAGS.VM = 0; it includes compatibility mode and 64-bit mode.
3. PVI mode and virtual-8086 mode each imply CPL = 3.

Operation

```
IF CR0.PE = 0  (* Executing in real-address mode *)
    THEN IF := 1; (* Set Interrupt Flag *)
    ELSE
        IF IOPL ≥ CPL  (* CPL = 3 if EFLAGS.VM = 1 *)
            THEN IF := 1; (* Set Interrupt Flag *)
            ELSE
                IF VME mode OR PVI mode
                    THEN
                        IF EFLAGS.VIP = 0
                            THEN VIF := 1; (* Set Virtual Interrupt Flag *)
                            ELSE #GP(0);
                        FI;
                    ELSE #GP(0);
                FI;
        FI;
FI;
```

Flags Affected

Either the IF flag or the VIF flag is set to 1. Other flags are unaffected.

Protected Mode Exceptions

#GP(0)	If CPL is greater than IOPL and PVI mode is not active.
	If CPL is greater than IOPL and EFLAGS.VIP = 1.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If IOPL is less than 3 and VME mode is not active.
	If IOPL is less than 3 and EFLAGS.VIP = 1.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

STMXCSR—Store MXCSR Register State

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F AE /3 STMXCSR *m32*	M	V/V	SSE	Store contents of MXCSR register to *m32*.
VEX.LZ.0F.WIG AE /3 VSTMXCSR *m32*	M	V/V	AVX	Store contents of MXCSR register to *m32*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	N/A	N/A	N/A

Description

Stores the contents of the MXCSR control and status register to the destination operand. The destination operand is a 32-bit memory location. The reserved bits in the MXCSR register are stored as 0s.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

VEX.L must be 0, otherwise instructions will #UD.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

```
m32 := MXCSR;
```

Intel C/C++ Compiler Intrinsic Equivalent

_mm_getcsr(void)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-22, "Type 5 Class Exception Conditions," additionally:

#UD	If VEX.L= 1, If VEX.vvvv ≠ 1111B.

STOS/STOSB/STOSW/STOSD/STOSQ—Store String

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
AA	STOS m8	ZO	Valid	Valid	For legacy mode, store AL at address ES:(E)DI; For 64-bit mode store AL at address RDI or EDI.
AB	STOS m16	ZO	Valid	Valid	For legacy mode, store AX at address ES:(E)DI; For 64-bit mode store AX at address RDI or EDI.
AB	STOS m32	ZO	Valid	Valid	For legacy mode, store EAX at address ES:(E)DI; For 64-bit mode store EAX at address RDI or EDI.
REX.W + AB	STOS m64	ZO	Valid	N.E.	Store RAX at address RDI or EDI.
AA	STOSB	ZO	Valid	Valid	For legacy mode, store AL at address ES:(E)DI; For 64-bit mode store AL at address RDI or EDI.
AB	STOSW	ZO	Valid	Valid	For legacy mode, store AX at address ES:(E)DI; For 64-bit mode store AX at address RDI or EDI.
AB	STOSD	ZO	Valid	Valid	For legacy mode, store EAX at address ES:(E)DI; For 64-bit mode store EAX at address RDI or EDI.
REX.W + AB	STOSQ	ZO	Valid	N.E.	Store RAX at address RDI or EDI.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

In non-64-bit and default 64-bit mode; stores a byte, word, or doubleword from the AL, AX, or EAX register (respectively) into the destination operand. The destination operand is a memory location, the address of which is read from either the ES:EDI or ES:DI register (depending on the address-size attribute of the instruction and the mode of operation). The ES segment cannot be overridden with a segment override prefix.

At the assembly-code level, two forms of the instruction are allowed: the "explicit-operands" form and the "no-operands" form. The explicit-operands form (specified with the STOS mnemonic) allows the destination operand to be specified explicitly. Here, the destination operand should be a symbol that indicates the size and location of the destination value. The source operand is then automatically selected to match the size of the destination operand (the AL register for byte operands, AX for word operands, EAX for doubleword operands). The explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the destination operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the ES:(E)DI register. These must be loaded correctly before the store string instruction is executed.

The no-operands form provides "short forms" of the byte, word, doubleword, and quadword versions of the STOS instructions. Here also ES:(E)DI is assumed to be the destination operand and AL, AX, or EAX is assumed to be the source operand. The size of the destination and source operands is selected by the mnemonic: STOSB (byte read from register AL), STOSW (word from AX), STOSD (doubleword from EAX).

After the byte, word, or doubleword is transferred from the register to the memory location, the (E)DI register is incremented or decremented according to the setting of the DF flag in the EFLAGS register. If the DF flag is 0, the register is incremented; if the DF flag is 1, the register is decremented (the register is incremented or decremented by 1 for byte operations, by 2 for word operations, by 4 for doubleword operations).

NOTE

To improve performance, more recent processors support modifications to the processor's operation during the string store operations initiated with STOS and STOSB. See Section 7.3.9.3 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for additional information on fast-string operation.

In 64-bit mode, the default address size is 64 bits, 32-bit address size is supported using the prefix 67H. Using a REX prefix in the form of REX.W promotes operation on doubleword operand to 64 bits. The promoted no-operand mnemonic is STOSQ. STOSQ (and its explicit operands variant) store a quadword from the RAX register into the destination addressed by RDI or EDI. See the summary chart at the beginning of this section for encoding data and limits.

The STOS, STOSB, STOSW, STOSD, STOSQ instructions can be preceded by the REP prefix for block stores of ECX bytes, words, or doublewords. More often, however, these instructions are used within a LOOP construct because data needs to be moved into the AL, AX, or EAX register before it can be stored. See "REP—Repeat String Operation (Prefix)" in this chapter for a description of the REP prefix.

Operation

Non-64-bit Mode:

```
IF (Byte store)
    THEN
        DEST := AL;
            THEN IF DF = 0
                THEN (E)DI := (E)DI + 1;
                ELSE (E)DI := (E)DI – 1;
            FI;
    ELSE IF (Word store)
        THEN
            DEST := AX;
                THEN IF DF = 0
                    THEN (E)DI := (E)DI + 2;
                    ELSE (E)DI := (E)DI – 2;
                FI;
        FI;
    ELSE IF (Doubleword store)
        THEN
            DEST := EAX;
                THEN IF DF = 0
                    THEN (E)DI := (E)DI + 4;
                    ELSE (E)DI := (E)DI – 4;
                FI;
        FI;
FI;
```

64-bit Mode:

```
IF (Byte store)
    THEN
        DEST := AL;
            THEN IF DF = 0
                THEN (R|E)DI := (R|E)DI + 1;
                ELSE (R|E)DI := (R|E)DI – 1;
            FI;
    ELSE IF (Word store)
        THEN
            DEST := AX;
                THEN IF DF = 0
                    THEN (R|E)DI := (R|E)DI + 2;
                    ELSE (R|E)DI := (R|E)DI – 2;
                FI;
        FI;
    ELSE IF (Doubleword store)
```

```
        THEN
            DEST := EAX;
                THEN IF DF = 0
                    THEN (R|E)DI := (R|E)DI + 4;
                    ELSE (R|E)DI := (R|E)DI – 4;
                FI;
        FI;
    ELSE IF (Quadword store using REX.W )
        THEN
            DEST := RAX;
                THEN IF DF = 0
                    THEN (R|E)DI := (R|E)DI + 8;
                    ELSE (R|E)DI := (R|E)DI – 8;
                FI;
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the limit of the ES segment.
	If the ES register contains a NULL segment selector.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the ES segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the ES segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

STR—Store Task Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 00 /1	STR r/m16	M	Valid	Valid	Stores segment selector from TR in r/m16.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	N/A	N/A	N/A

Description

Stores the segment selector from the task register (TR) in the destination operand. The destination operand can be a general-purpose register or a memory location. The segment selector stored with this instruction points to the task state segment (TSS) for the currently running task.

When the destination operand is a 32-bit register, the 16-bit segment selector is copied into the lower 16 bits of the register and the upper 16 bits of the register are cleared. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of operand size.

In 64-bit mode, operation is the same. The size of the memory operand is fixed at 16 bits. In register stores, the 2-byte TR is zero extended if stored to a 64-bit register.

The STR instruction is useful only in operating-system software. It can only be executed in protected mode.

Operation

```
DEST := TR(SegmentSelector);
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is a memory operand that is located in a non-writable segment or if the effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If CR4.UMIP = 1 and CPL > 0.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	The STR instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The STR instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form. If CR4.UMIP = 1 and CPL > 0.
#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

STTILECFG—Store Tile Configuration

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 49 !(11):000:bbb STTILECFG m512	A	V/N.E.	AMX_TILE	Store tile configuration in m512.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (w)	N/A	N/A	N/A

Description

The STTILECFG instruction takes a pointer to a 64-byte memory location (described in Table 3-56 in the "LDTILECFG—Load Tile Configuration" entry) that will, after successful execution of this instruction, contain the description of the tiles that were configured. In order to configure tiles, the AMX_TILE bit in CPUID must be set and the operating system has to have enabled the tiles architecture.

If the tiles are not configured, then STTILECFG stores 64B of zeros to the indicated memory location.

Any attempt to execute the STTILECFG instruction inside an Intel TSX transaction will result in a transaction abort.

Operation

STTILECFG mem

```
if TILES_CONFIGURED == 0:
    //write 64 bytes of zeros at mem pointer
    buf[0..63] := 0
    write_memory(mem, 64, buf)
else:
    buf.byte[0] := tilecfg.palette_id
    buf.byte[1] := tilecfg.start_row
    buf.byte[2..15] := 0

    p := 16
    for n in 0 ... palette_table[tilecfg.palette_id].max_names-1:
        buf.word[p/2] := tilecfg.t[n].colsb
        p := p + 2
    if p < 47:
        buf.byte[p..47] := 0
    p := 48
    for n in 0 ... palette_table[tilecfg.palette_id].max_names-1:
        buf.byte[p++] := tilecfg.t[n].rows
    if p < 63:
        buf.byte[p..63] := 0

    write_memory(mem, 64, buf)
```

Intel C/C++ Compiler Intrinsic Equivalent

STTILECFGvoid _tile_storeconfig(void *);

Flags Affected

None.

Exceptions

AMX-E2; see Section 2.10, "Intel® AMX Instruction Exception Classes," for details.

STUI—Set User Interrupt Flag

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 01 EF STUI	ZO	V/I	UINTR	Set user interrupt flag.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A	N/A

Description

STUI sets the user interrupt flag (UIF). Its effect takes place immediately; a user interrupt may be delivered on the instruction boundary following STUI. (This is in contrast with STI, whose effect is delayed by one instruction).

An execution of STUI inside a transactional region causes a transactional abort; the abort loads EAX as it would have had it been due to an execution of STI.

Operation

```
UIF := 1;
```

Flags Affected

None.

Protected Mode Exceptions

#UD	The STUI instruction is not recognized in protected mode.

Real-Address Mode Exceptions

#UD	The STUI instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The STUI instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	The STUI instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If executed inside an enclave.
	If CR4.UINTR = 0.
	If CPUID.07H.00H:EDX.UINTR[5] = 0.

SUB—Subtract

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
2C ib	SUB AL, imm8	I	Valid	Valid	Subtract imm8 from AL.
2D iw	SUB AX, imm16	I	Valid	Valid	Subtract imm16 from AX.
2D id	SUB EAX, imm32	I	Valid	Valid	Subtract imm32 from EAX.
REX.W + 2D id	SUB RAX, imm32	I	Valid	N.E.	Subtract imm32 sign-extended to 64-bits from RAX.
80 /5 ib	SUB r/m8[1], imm8	MI	Valid	Valid	Subtract imm8 from r/m8.
81 /5 iw	SUB r/m16, imm16	MI	Valid	Valid	Subtract imm16 from r/m16.
81 /5 id	SUB r/m32, imm32	MI	Valid	Valid	Subtract imm32 from r/m32.
REX.W + 81 /5 id	SUB r/m64, imm32	MI	Valid	N.E.	Subtract imm32 sign-extended to 64-bits from r/m64.
83 /5 ib	SUB r/m16, imm8	MI	Valid	Valid	Subtract sign-extended imm8 from r/m16.
83 /5 ib	SUB r/m32, imm8	MI	Valid	Valid	Subtract sign-extended imm8 from r/m32.
REX.W + 83 /5 ib	SUB r/m64, imm8	MI	Valid	N.E.	Subtract sign-extended imm8 from r/m64.
28 /r	SUB r/m8[1], r8[1]	MR	Valid	Valid	Subtract r8 from r/m8.
29 /r	SUB r/m16, r16	MR	Valid	Valid	Subtract r16 from r/m16.
29 /r	SUB r/m32, r32	MR	Valid	Valid	Subtract r32 from r/m32.
REX.W + 29 /r	SUB r/m64, r64	MR	Valid	N.E.	Subtract r64 from r/m64.
2A /r	SUB r8[1], r/m8[1]	RM	Valid	Valid	Subtract r/m8 from r8.
2B /r	SUB r16, r/m16	RM	Valid	Valid	Subtract r/m16 from r16.
2B /r	SUB r32, r/m32	RM	Valid	Valid	Subtract r/m32 from r32.
REX.W + 2B /r	SUB r64, r/m64	RM	Valid	N.E.	Subtract r/m64 from r64.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	imm8/16/32	N/A	N/A
MI	ModRM:r/m (r, w)	imm8/16/32	N/A	N/A
MR	ModRM:r/m (r, w)	ModRM:reg (r)	N/A	N/A
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A

Description

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, register, or memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The SUB instruction performs integer subtraction. It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate an overflow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

Operation

```
DEST := (DEST – SRC);
```

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

SUBPD—Subtract Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5C /r SUBPD xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed double precision floating-point values in xmm2/mem from xmm1 and store result in xmm1.
VEX.128.66.0F.WIG 5C /r VSUBPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Subtract packed double precision floating-point values in xmm3/mem from xmm2 and store result in xmm1.
VEX.256.66.0F.WIG 5C /r VSUBPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Subtract packed double precision floating-point values in ymm3/mem from ymm2 and store result in ymm1.
EVEX.128.66.0F.W1 5C /r VSUBPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Subtract packed double precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1 with writemask k1.
EVEX.256.66.0F.W1 5C /r VSUBPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Subtract packed double precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1 with writemask k1.
EVEX.512.66.0F.W1 5C /r VSUBPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	C	V/V	AVX512F OR AVX10.1	Subtract packed double precision floating-point values from zmm3/m512/m64bcst to zmm2 and store result in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD subtract of the two, four or eight packed double precision floating-point values of the second Source operand from the first Source operand, and stores the packed double precision floating-point results in the destination operand.

VEX.128 and EVEX.128 encoded versions: The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX.512 encoded version: The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The first source operand and destination operands are ZMM registers. The destination operand is conditionally updated according to the writemask.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VSUBPD (EVEX Encoded Versions When SRC2 Operand is a Vector Register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC1[i+63:i] - SRC2[i+63:i]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63:0] := 0
        FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VSUBPD (EVEX Encoded Versions When SRC2 Operand is a Memory Source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1)
                THEN    DEST[i+63:i] := SRC1[i+63:i] - SRC2[63:0];
                ELSE    EST[i+63:i] := SRC1[i+63:i] - SRC2[i+63:i];
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63:0] := 0
        FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VSUBPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0] - SRC2[63:0]
DEST[127:64] := SRC1[127:64] - SRC2[127:64]
DEST[191:128] := SRC1[191:128] - SRC2[191:128]
DEST[255:192] := SRC1[255:192] - SRC2[255:192]
DEST[MAXVL-1:256] := 0
```

VSUBPD (VEX.128 Encoded Version)
DEST[63:0] := SRC1[63:0] - SRC2[63:0]
DEST[127:64] := SRC1[127:64] - SRC2[127:64]
DEST[MAXVL-1:128] := 0

SUBPD (128-bit Legacy SSE Version)
DEST[63:0] := DEST[63:0] - SRC[63:0]
DEST[127:64] := DEST[127:64] - SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VSUBPD __m512d _mm512_sub_pd (__m512d a, __m512d b);
VSUBPD __m512d _mm512_mask_sub_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);
VSUBPD __m512d _mm512_maskz_sub_pd (__mmask8 k, __m512d a, __m512d b);
VSUBPD __m512d _mm512_sub_round_pd (__m512d a, __m512d b, int);
VSUBPD __m512d _mm512_mask_sub_round_pd (__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VSUBPD __m512d _mm512_maskz_sub_round_pd (__mmask8 k, __m512d a, __m512d b, int);
VSUBPD __m256d _mm256_sub_pd (__m256d a, __m256d b);
VSUBPD __m256d _mm256_mask_sub_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);
VSUBPD __m256d _mm256_maskz_sub_pd (__mmask8 k, __m256d a, __m256d b);
SUBPD __m128d _mm_sub_pd (__m128d a, __m128d b);
VSUBPD __m128d _mm_mask_sub_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VSUBPD __m128d _mm_maskz_sub_pd (__mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

SUBPS—Subtract Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5C /r SUBPS xmm1, xmm2/m128	A	V/V	SSE	Subtract packed single precision floating-point values in xmm2/mem from xmm1 and store result in xmm1.
VEX.128.0F.WIG 5C /r VSUBPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Subtract packed single precision floating-point values in xmm3/mem from xmm2 and stores result in xmm1.
VEX.256.0F.WIG 5C /r VSUBPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Subtract packed single precision floating-point values in ymm3/mem from ymm2 and stores result in ymm1.
EVEX.128.0F.W0 5C /r VSUBPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Subtract packed single precision floating-point values from xmm3/m128/m32bcst to xmm2 and stores result in xmm1 with writemask k1.
EVEX.256.0F.W0 5C /r VSUBPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Subtract packed single precision floating-point values from ymm3/m256/m32bcst to ymm2 and stores result in ymm1 with writemask k1.
EVEX.512.0F.W0 5C /r VSUBPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	C	V/V	AVX512F OR AVX10.1	Subtract packed single precision floating-point values in zmm3/m512/m32bcst from zmm2 and stores result in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD subtract of the packed single precision floating-point values in the second Source operand from the First Source operand, and stores the packed single precision floating-point results in the destination operand.

VEX.128 and EVEX.128 encoded versions: The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX.512 encoded version: The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The first source operand and destination operands are ZMM registers. The destination operand is conditionally updated according to the writemask.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VSUBPS (EVEX Encoded Versions When SRC2 Operand is a Vector Register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC1[i+31:i] - SRC2[i+31:i]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VSUBPS (EVEX Encoded Versions When SRC2 Operand is a Memory Source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1)
                THEN    DEST[i+31:i] := SRC1[i+31:i] - SRC2[31:0];
                ELSE    DEST[i+31:i] := SRC1[i+31:i] - SRC2[i+31:i];
            FI;

    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VSUBPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[31:0] - SRC2[31:0]
DEST[63:32] := SRC1[63:32] - SRC2[63:32]
DEST[95:64] := SRC1[95:64] - SRC2[95:64]
DEST[127:96] := SRC1[127:96] - SRC2[127:96]
DEST[159:128] := SRC1[159:128] - SRC2[159:128]
DEST[191:160] := SRC1[191:160] - SRC2[191:160]
DEST[223:192] := SRC1[223:192] - SRC2[223:192]
DEST[255:224] := SRC1[255:224] - SRC2[255:224].
DEST[MAXVL-1:256] := 0
```

VSUBPS (VEX.128 Encoded Version)
DEST[31:0] := SRC1[31:0] - SRC2[31:0]
DEST[63:32] := SRC1[63:32] - SRC2[63:32]
DEST[95:64] := SRC1[95:64] - SRC2[95:64]
DEST[127:96] := SRC1[127:96] - SRC2[127:96]
DEST[MAXVL-1:128] := 0

SUBPS (128-bit Legacy SSE Version)
DEST[31:0] := SRC1[31:0] - SRC2[31:0]
DEST[63:32] := SRC1[63:32] - SRC2[63:32]
DEST[95:64] := SRC1[95:64] - SRC2[95:64]
DEST[127:96] := SRC1[127:96] - SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VSUBPS __m512 _mm512_sub_ps (__m512 a, __m512 b);
VSUBPS __m512 _mm512_mask_sub_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VSUBPS __m512 _mm512_maskz_sub_ps (__mmask16 k, __m512 a, __m512 b);
VSUBPS __m512 _mm512_sub_round_ps (__m512 a, __m512 b, int);
VSUBPS __m512 _mm512_mask_sub_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VSUBPS __m512 _mm512_maskz_sub_round_ps (__mmask16 k, __m512 a, __m512 b, int);
VSUBPS __m256 _mm256_sub_ps (__m256 a, __m256 b);
VSUBPS __m256 _mm256_mask_sub_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VSUBPS __m256 _mm256_maskz_sub_ps (__mmask16 k, __m256 a, __m256 b);
SUBPS __m128 _mm_sub_ps (__m128 a, __m128 b);
VSUBPS __m128 _mm_mask_sub_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VSUBPS __m128 _mm_maskz_sub_ps (__mmask16 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, "Type 2 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

SUBSD—Subtract Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5C /r SUBSD xmm1, xmm2/m64	A	V/V	SSE2	Subtract the low double precision floating-point value in xmm2/m64 from xmm1 and store the result in xmm1.
VEX.LIG.F2.0F.WIG 5C /r VSUBSD xmm1,xmm2, xmm3/m64	B	V/V	AVX	Subtract the low double precision floating-point value in xmm3/m64 from xmm2 and store the result in xmm1.
EVEX.LLIG.F2.0F.W1 5C /r VSUBSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F OR AVX10.1	Subtract the low double precision floating-point value in xmm3/m64 from xmm2 and store the result in xmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Subtract the low double precision floating-point value in the second source operand from the first source operand and stores the double precision floating-point result in the low quadword of the destination operand.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the write-mask.

Software should ensure VSUBSD is encoded with VEX.L=0. Encoding VSUBSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VSUBSD (EVEX Encoded Version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := SRC1[63:0] - SRC2[63:0]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VSUBSD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] - SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

SUBSD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] - SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VSUBSD __m128d _mm_mask_sub_sd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VSUBSD __m128d _mm_maskz_sub_sd (__mmask8 k, __m128d a, __m128d b);
VSUBSD __m128d _mm_sub_round_sd (__m128d a, __m128d b, int);
VSUBSD __m128d _mm_mask_sub_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSUBSD __m128d _mm_maskz_sub_round_sd (__mmask8 k, __m128d a, __m128d b, int);
SUBSD __m128d _mm_sub_sd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-49, "Type E3 Class Exception Conditions."

SUBSS—Subtract Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5C /r SUBSS xmm1, xmm2/m32	A	V/V	SSE	Subtract the low single precision floating-point value in xmm2/m32 from xmm1 and store the result in xmm1.
VEX.LIG.F3.0F.WIG 5C /r VSUBSS xmm1,xmm2, xmm3/m32	B	V/V	AVX	Subtract the low single precision floating-point value in xmm3/m32 from xmm2 and store the result in xmm1.
EVEX.LLIG.F3.0F.W0 5C /r VSUBSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	C	V/V	AVX512F OR AVX10.1	Subtract the low single precision floating-point value in xmm3/m32 from xmm2 and store the result in xmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Subtract the low single precision floating-point value from the second source operand and the first source operand and store the double precision floating-point result in the low doubleword of the destination operand.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL-1:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the write-mask.

Software should ensure VSUBSS is encoded with VEX.L=0. Encoding VSUBSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VSUBSS (EVEX Encoded Version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := SRC1[31:0] - SRC2[31:0]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VSUBSS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] - SRC2[31:0]
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

SUBSS (128-bit Legacy SSE Version)

```
DEST[31:0] := DEST[31:0] - SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VSUBSS __m128 _mm_mask_sub_ss (__m128 s, __mmask8 k, __m128 a, __m128 b);
VSUBSS __m128 _mm_maskz_sub_ss (__mmask8 k, __m128 a, __m128 b);
VSUBSS __m128 _mm_sub_round_ss (__m128 a, __m128 b, int);
VSUBSS __m128 _mm_mask_sub_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSUBSS __m128 _mm_maskz_sub_round_ss (__mmask8 k, __m128 a, __m128 b, int);
SUBSS __m128 _mm_sub_ss (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-49, "Type E3 Class Exception Conditions."

SWAPGS—Swap GS Base Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 F8	SWAPGS	ZO	Valid	Invalid	Exchanges the current GS base register value with the IA32_KERNEL_GS_BASE MSR.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

SWAPGS exchanges the current GS base register value with the IA32_KERNEL_GS_BASE MSR (MSR address C0000102H). The SWAPGS instruction is a privileged instruction intended for use by system software.

When using SYSCALL to implement system calls, there is no kernel stack at the OS entry point. Neither is there a straightforward method to obtain a pointer to kernel structures from which the kernel stack pointer could be read. Thus, the kernel cannot save general purpose registers or reference memory.

By design, SWAPGS does not require any general purpose registers or memory operands. No registers need to be saved before using the instruction. SWAPGS exchanges the CPL 0 data pointer from the IA32_KERNEL_GS_BASE MSR with the GS base register. The kernel can then use the GS prefix on normal memory references to access kernel data structures. Similarly, when the OS kernel is entered using an interrupt or exception (where the kernel stack is already set up), SWAPGS can be used to quickly get a pointer to the kernel data structures.

The IA32_KERNEL_GS_BASE MSR itself is only accessible using RDMSR/WRMSR instructions. Those instructions are only accessible at privilege level 0. The WRMSR instruction ensures that the IA32_KERNEL_GS_BASE MSR contains a canonical address.

The instruction cannot be executed when FRED transitions are enabled. FRED transitions perform the same swap when changing the CPL.

Operation

```
IF CS.L ≠ 1 (* Not in 64-Bit Mode *) OR CR4.FRED = 1
    THEN
        #UD; FI;
IF CPL ≠ 0
    THEN #GP(0); FI;
tmp := GS.base;
GS.base := IA32_KERNEL_GS_BASE;
IA32_KERNEL_GS_BASE := tmp;
```

Flags Affected

None.

Protected Mode Exceptions

#UD	The SWAPGS instruction is not recognized in protected mode.

Real-Address Mode Exceptions

#UD	The SWAPGS instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The SWAPGS instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	The SWAPGS instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#GP(0)	If CPL ≠ 0.
#UD	If the LOCK prefix is used.
	If CR4.FRED = 1.

SYSCALL—Fast System Call

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 05	SYSCALL	ZO	Valid	Invalid	Fast call to privilege level 0 system procedures.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

SYSCALL invokes an OS system-call handler at privilege level 0. Its operation depends on whether FRED transitions are enabled.

Operation When FRED Transitions Are Not Enabled

When FRED transitions are not enabled, SYSCALL invokes the OS handler by loading RIP from the IA32_LSTAR MSR (after saving the address of the instruction following SYSCALL into RCX). (MSR writes ensure that the IA32_LSTAR MSR always contain a canonical address.) The OS handler returns using the SYSRET instruction.

SYSCALL also saves RFLAGS into R11 and then masks RFLAGS using the IA32_FMASK MSR (MSR address C0000084H); specifically, the processor clears in RFLAGS every bit corresponding to a bit that is set in the IA32_FMASK MSR.

SYSCALL loads the CS and SS selectors with values derived from bits 47:32 of the IA32_STAR MSR. However, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSCALL instruction does not ensure this correspondence.

The SYSCALL instruction does not save the stack pointer (RSP). If the OS system-call handler will change the stack pointer, it is the responsibility of software to save the previous value of the stack pointer. This might be done prior to executing SYSCALL, with software restoring the stack pointer with the instruction following SYSCALL (which will be executed after SYSRET). Alternatively, the OS system-call handler may save the stack pointer and restore it before executing SYSRET.

When shadow stacks are enabled at a privilege level where the SYSCALL instruction is invoked, the SSP is saved to the IA32_PL3_SSP MSR. If shadow stacks are enabled at privilege level 0, the SSP is loaded with 0. Refer to Chapter 6, "Procedure Calls, Interrupts, and Exceptions," and Chapter 18, "Control-flow Enforcement Technology (CET)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for additional CET details.

When FRED Transitions are Enabled

When FRED transitions are enabled, SYSCALL invokes the OS handler by performing FRED event delivery. See Section 8.3, "FRED Event Delivery," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3. The event is delivered using event type 7 and vector 1. With FRED transitions, the OS handler uses the ERETU instruction to return to calling code operating at CPL 3.

Instruction ordering. Instructions following a SYSCALL may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the SYSCALL have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

Operation

```
IF IA32_EFER.LMA = 0 OR CS.L = 0    (* SYSCALL can be used only in 64-bit mode *)
    THEN #UD;
ELSE IF CR4.FRED = 0
```

```
    THEN
        IF IA32_EFER.SCE = 0
            THEN #UD;
            ELSE
                RCX := RIP;                             (* Will contain address of next instruction *)
                RIP := IA32_LSTAR;
                R11 := RFLAGS;
                RFLAGS := RFLAGS AND NOT(IA32_FMASK);
                CS.Selector := IA32_STAR[47:32] AND FFFCH    (* Operating system provides CS; RPL forced to 0 *)
                (* Set rest of CS to a fixed value *)
                CS.Base := 0;                           (* Flat segment *)
                CS.Limit := FFFFFH;                     (* With 4-KByte granularity, implies a 4-GByte limit *)
                CS.Type := 11;                          (* Execute/read code, accessed *)
                CS.S := 1;
                CS.DPL := 0;
                CS.P := 1;
                CS.L := 1;                              (* Entry is to 64-bit mode *)
                CS.D := 0;                              (* Required if CS.L = 1 *)
                CS.G := 1;                              (* 4-KByte granularity *)
                IF ShadowStackEnabled(CPL)
                    THEN (* adjust so bits 63:N get the value of bit N-1, where N is the CPU's maximum linear-address width *)
                        IA32_PL3_SSP := LA_adjust(SSP);
                        (* With shadow stacks enabled the system call is supported from Ring 3 to Ring 0 *)
                        (* OS supporting Ring 0 to Ring 0 system calls or Ring 1/2 to ring 0 system call *)
                        (* Must preserve the contents of IA32_PL3_SSP to avoid losing ring 3 state *)
                FI;

                CPL := 0;

                IF ShadowStackEnabled(CPL)
                    SSP := 0;
                FI;
                IF EndbranchEnabled(CPL)
                    IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                    IA32_S_CET.SUPPRESS = 0
                FI;
                SS.Selector := IA32_STAR[47:32] + 8;    (* SS just above CS *)
                (* Set rest of SS to a fixed value *)
                SS.Base := 0;                           (* Flat segment *)
                SS.Limit := FFFFFH;                     (* With 4-KByte granularity, implies a 4-GByte limit *)
                SS.Type := 3;                           (* Read/write data, accessed *)
                SS.S := 1;
                SS.DPL := 0;
                SS.P := 1;
                SS.B := 1;                              (* 32-bit stack segment *)
                SS.G := 1;                              (* 4-KByte granularity *)
        FI;
    ELSE (* CR4.FRED = 1 *)
        FRED event delivery of SYSCALL;                 (* save instruction length on stack *)
FI;
```

Flags Affected

All.

Protected Mode Exceptions

#UD	The SYSCALL instruction is not recognized in protected mode.

Real-Address Mode Exceptions

#UD	The SYSCALL instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The SYSCALL instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	The SYSCALL instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD	If IA32_EFER.SCE = 0.
	If the LOCK prefix is used.
#GP(0)	If FRED event delivery of SYSCALL would load RIP with a non-canonical address.
	If FRED event delivery of SYSCALL would load SSP with an address that is not 8-byte aligned.
	If a shadow-stack access performed by FRED event delivery of SYSCALL uses a non-canonical address or causes a LASS violation.
#SS(0)	If an ordinary stack access performed by FRED event delivery of SYSCALL uses a non-canonical address or causes a LASS violation.
#PF	If a memory access performed by FRED event delivery of SYSCALL causes a page fault.

SYSENTER—Fast System Call

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 34	SYSENTER	ZO	Valid	Valid	Fast call to privilege level 0 system procedures.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

SYSENTER invokes an OS system-call handler at privilege level 0. Its operation depends on whether FRED transitions are enabled.

Operation When FRED Transitions Are Not Enabled

When FRED transitions are not enabled, SYSENTER is a companion instruction to SYSEXIT. SYSENTER is optimized to provide the maximum performance for system calls from user code running at privilege level 3 to operating system or executive procedures running at privilege level 0.

When executed in IA-32e mode, the SYSENTER instruction transitions the logical processor to 64-bit mode; otherwise, the logical processor remains in protected mode.

Prior to executing the SYSENTER instruction, software must specify the privilege level 0 code segment and code entry point, and the privilege level 0 stack segment and stack pointer by writing values to the following MSRs:

- **IA32_SYSENTER_CS** (MSR address 174H) — The lower 16 bits of this MSR are the segment selector for the privilege level 0 code segment. This value is also used to determine the segment selector of the privilege level 0 stack segment (see the Operation section). This value cannot indicate a null selector.
- **IA32_SYSENTER_EIP** (MSR address 176H) — The value of this MSR is loaded into RIP (thus, this value references the first instruction of the selected operating procedure or routine). In protected mode, only bits 31:0 are loaded.
- **IA32_SYSENTER_ESP** (MSR address 175H) — The value of this MSR is loaded into RSP (thus, this value contains the stack pointer for the privilege level 0 stack). This value cannot represent a non-canonical address. In protected mode, only bits 31:0 are loaded.

MSR writes ensure that the IA32_SYSENTER_EIP and IA32_SYSENTER_ESP MSRs always contain canonical addresses.

While SYSENTER loads the CS and SS selectors with values derived from the IA32_SYSENTER_CS MSR, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSENTER instruction does not ensure this correspondence.

The SYSENTER instruction cannot be invoked from real-address mode.

The SYSENTER and SYSEXIT instructions are companion instructions, but they do not constitute a call/return pair. When executing a SYSENTER instruction, the processor does not save state information for the user code (e.g., the instruction pointer), and neither the SYSENTER nor the SYSEXIT instruction supports passing parameters on the stack.

To use the SYSENTER and SYSEXIT instructions as companion instructions for transitions between privilege level 3 code and privilege level 0 operating system procedures, the following conventions must be followed:

- The segment descriptors for the privilege level 0 code and stack segments and for the privilege level 3 code and stack segments must be contiguous in a descriptor table. This convention allows the processor to compute the segment selectors from the value entered in the SYSENTER_CS_MSR MSR.
- The fast system call "stub" routines executed by user code (typically in shared libraries or DLLs) must save the required return IP and processor state information if a return to the calling procedure is required. Likewise, the

operating system or executive procedures called with SYSENTER instructions must have access to and use this saved return and state information when returning to the user code.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

```
IF CPUID SEP bit is set
    THEN IF (Family = 6) and (Model < 3) and (Stepping < 3)
        THEN
            SYSENTER/SYSEXIT_Not_Supported; FI;
        ELSE
            SYSENTER/SYSEXIT_Supported; FI;
FI;
```

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns a the SEP flag as set, but does not support the SYSENTER/SYSEXIT instructions.

When shadow stacks are enabled at privilege level where SYSENTER instruction is invoked, the SSP is saved to the IA32_PL3_SSP MSR. If shadow stacks are enabled at privilege level 0, the SSP is loaded with 0. Refer to Chapter 6, "Procedure Calls, Interrupts, and Exceptions," and Chapter 18, "Control-flow Enforcement Technology (CET)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for additional CET details.

When FRED Transitions are Enabled

When FRED transitions are enabled, SYSENTER invokes the OS handler by performing FRED event delivery. See Section 8.3, "FRED Event Delivery," in Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3. The event is delivered using event type 7 and vector 2. With FRED transitions, the OS handler uses the ERETU instruction to return to calling code operating at CPL 3.

Instruction ordering. Instructions following a SYSENTER may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the SYSENTER have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

Operation

```
IF CR0.PE = 0 OR (CR4.FRED = 0 AND IA32_SYSENTER_CS[15:2] = 0)
    THEN #GP(0); FI;

IF CR4.FRED = 0
    THEN
        RFLAGS.VM := 0;                         (* Ensures protected mode execution *)
        RFLAGS.IF := 0;                         (* Mask interrupts *)
        IF in IA-32e mode
            THEN
                RSP := IA32_SYSENTER_ESP;
                RIP := IA32_SYSENTER_EIP;
        ELSE
                ESP := IA32_SYSENTER_ESP[31:0];
                EIP := IA32_SYSENTER_EIP[31:0];
        FI;
        CS.Selector := IA32_SYSENTER_CS[15:0] AND FFFCH;
                                        (* Operating system provides CS; RPL forced to 0 *)
        (* Set rest of CS to a fixed value *)
        CS.Base := 0;                           (* Flat segment *)
        CS.Limit := FFFFFH;                     (* With 4-KByte granularity, implies a 4-GByte limit *)
        CS.Type := 11;                          (* Execute/read code, accessed *)
```

```
        CS.S := 1;
        CS.DPL := 0;
        CS.P := 1;
        IF in IA-32e mode
            THEN
                CS.L := 1;                          (* Entry is to 64-bit mode *)
                CS.D := 0;                          (* Required if CS.L = 1 *)
            ELSE
                CS.L := 0;
                CS.D := 1;                          (* 32-bit code segment*)
        FI;
        CS.G := 1;                                  (* 4-KByte granularity *)

        IF ShadowStackEnabled(CPL)
            THEN
                IF IA32_EFER.LMA = 0
                    THEN IA32_PL3_SSP := SSP;
                    ELSE (* adjust so bits 63:N get the value of bit N-1, where N is the CPU's maximum linear-address width *)
                        IA32_PL3_SSP := LA_adjust(SSP);
                FI;
        FI;

        CPL := 0;

        IF ShadowStackEnabled(CPL)
            SSP := 0;
        FI;
        IF EndbranchEnabled(CPL)
            IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
            IA32_S_CET.SUPPRESS = 0
        FI;
        SS.Selector := CS.Selector + 8;             (* SS just above CS *)
        (* Set rest of SS to a fixed value *)
        SS.Base := 0;                               (* Flat segment *)
        SS.Limit := FFFFFH;                         (* With 4-KByte granularity, implies a 4-GByte limit *)
        SS.Type := 3;                               (* Read/write data, accessed *)
        SS.S := 1;
        SS.DPL := 0;
        SS.P := 1;
        SS.B := 1;                                  (* 32-bit stack segment*)
        SS.G := 1;                                  (* 4-KByte granularity *)
    ELSE (* CR4.FRED = 1 *)
        FRED event delivery of SYSENTER;            (* save instruction length on stack *)
FI;
```

Flags Affected

VM, IF (see Operation above).

Protected Mode Exceptions

#GP(0)	If IA32_SYSENTER_CS[15:2] = 0.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	The SYSENTER instruction is not recognized in real-address mode.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.
#GP(0)	If CR4.FRED = 0 and IA32_SYSENTER_CS[15:2] = 0.
	If FRED event delivery of SYSENTER would load RIP with a non-canonical address.
	If FRED event delivery of SYSENTER would load SSP with an address that is not 8-byte aligned.
	If a shadow-stack access performed by FRED event delivery of SYSENTER uses a non-canonical address or causes a LASS violation.
#SS(0)	If an ordinary stack access performed by FRED event delivery of SYSENTER uses a non-canonical address or causes a LASS violation.
#PF	If a memory access performed by FRED event delivery of SYSENTER causes a page fault.

64-Bit Mode Exceptions

Same exceptions as in compatibility mode.

SYSEXIT—Fast Return from Fast System Call

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 35	SYSEXIT	ZO	Valid	Valid	Fast return to privilege level 3 user code.
REX.W + 0F 35	SYSEXIT	ZO	Valid	Valid	Fast return to 64-bit mode privilege level 3 user code.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Executes a fast return to privilege level 3 user code. SYSEXIT is a companion instruction to the SYSENTER instruction. The instruction is optimized to provide the maximum performance for returns from system procedures executing at protections levels 0 to user procedures executing at protection level 3. It must be executed from code executing at privilege level 0.

With a 64-bit operand size, SYSEXIT remains in 64-bit mode; otherwise, it either enters compatibility mode (if the logical processor is in IA-32e mode) or remains in protected mode (if it is not).

Prior to executing SYSEXIT, software must specify the privilege level 3 code segment and code entry point, and the privilege level 3 stack segment and stack pointer by writing values into the following MSR and general-purpose registers:

- **IA32_SYSENTER_CS** (MSR address 174H) — Contains a 32-bit value that is used to determine the segment selectors for the privilege level 3 code and stack segments (see the Operation section)
- **RDX** — The canonical address in this register is loaded into RIP (thus, this value references the first instruction to be executed in the user code). If the return is not to 64-bit mode, only bits 31:0 are loaded.
- **RCX** — The canonical address in this register is loaded into RSP (thus, this value contains the stack pointer for the privilege level 3 stack). If the return is not to 64-bit mode, only bits 31:0 are loaded.

The IA32_SYSENTER_CS MSR can be read from and written to using RDMSR and WRMSR.

While SYSEXIT loads the CS and SS selectors with values derived from the IA32_SYSENTER_CS MSR, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSEXIT instruction does not ensure this correspondence.

The SYSEXIT instruction can be invoked from all operating modes except real-address mode and virtual-8086 mode.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

```
IF CPUID SEP bit is set
    THEN IF (Family = 6) and (Model < 3) and (Stepping < 3)
        THEN
            SYSENTER/SYSEXIT_Not_Supported; FI;
        ELSE
            SYSENTER/SYSEXIT_Supported; FI;
FI;
```

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns a the SEP flag as set, but does not support the SYSENTER/SYSEXIT instructions.

When shadow stacks are enabled at privilege level 3 the instruction loads SSP with value from IA32_PL3_SSP MSR. Refer to Chapter 7, "Interrupt and Exception Handling," and Chapter 18, "Control-flow Enforcement Technology (CET)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for additional CET details.

The instruction cannot be executed when FRED transitions are enabled. An operating system that has enabled FRED transitions should use ERETU instead.

Instruction ordering. Instructions following a SYSEXIT may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the SYSEXIT have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

Operation

```
IF CR4.FRED = 1
    THEN #UD; FI;
IF IA32_SYSENTER_CS[15:2] = 0 OR CR0.PE = 0 OR CPL ≠ 0 THEN #GP(0); FI;
IF operand size is 64-bit
    THEN    (* Return to 64-bit mode *)
        RSP := RCX;
        RIP := RDX;
    ELSE    (* Return to protected mode or compatibility mode *)
        RSP := ECX;
        RIP := EDX;
FI;
IF operand size is 64-bit                   (* Operating system provides CS; RPL forced to 3 *)
    THEN CS.Selector := IA32_SYSENTER_CS[15:0] + 32;
    ELSE CS.Selector := IA32_SYSENTER_CS[15:0] + 16;
FI;
CS.Selector := CS.Selector OR 3;            (* RPL forced to 3 *)
(* Set rest of CS to a fixed value *)
CS.Base := 0;                               (* Flat segment *)
CS.Limit := FFFFFH;                         (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type := 11;                              (* Execute/read code, accessed *)
CS.S := 1;
CS.DPL := 3;
CS.P := 1;
IF operand size is 64-bit
    THEN    (* return to 64-bit mode *)
        CS.L := 1;                          (* 64-bit code segment *)
        CS.D := 0;                          (* Required if CS.L = 1 *)
    ELSE    (* return to protected mode or compatibility mode *)
        CS.L := 0;
        CS.D := 1;                          (* 32-bit code segment*)
FI;
CS.G := 1;                                  (* 4-KByte granularity *)
CPL := 3;

IF ShadowStackEnabled(CPL)
    THEN SSP := IA32_PL3_SSP;
FI;
SS.Selector := CS.Selector + 8;             (* SS just above CS *)
(* Set rest of SS to a fixed value *)
SS.Base := 0;                               (* Flat segment *)
SS.Limit := FFFFFH;                         (* With 4-KByte granularity, implies a 4-GByte limit *)
```

```
SS.Type := 3;                   (* Read/write data, accessed *)
SS.S := 1;
SS.DPL := 3;
SS.P := 1;
SS.B := 1;                      (* 32-bit stack segment*)
SS.G := 1;                      (* 4-KByte granularity *)
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If IA32_SYSENTER_CS[15:2] = 0.
	If CPL ≠ 0.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	The SYSEXIT instruction is not recognized in real-address mode.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	The SYSEXIT instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#GP(0)	If IA32_SYSENTER_CS[15:2] = 0.
	If CPL ≠ 0.
#UD	If the LOCK prefix is used.
	If CR4.FRED = 1.

64-Bit Mode Exceptions

#GP(0)	If IA32_SYSENTER_CS = 0.
	If CPL ≠ 0.
	If RCX or RDX contains a non-canonical address.
#UD	If the LOCK prefix is used.
	If CR4.FRED = 1.

SYSRET—Return From Fast System Call

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 07	SYSRET	ZO	Valid	Invalid	Return to compatibility mode from fast system call.
REX.W + 0F 07	SYSRET	ZO	Valid	Invalid	Return to 64-bit mode from fast system call.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

SYSRET is a companion instruction to the SYSCALL instruction. It returns from an OS system-call handler to user code at privilege level 3. It does so by loading RIP from RCX and loading RFLAGS from R11.[1] With a 64-bit operand size, SYSRET remains in 64-bit mode; otherwise, it enters compatibility mode and only the low 32 bits of the registers are loaded.

SYSRET loads the CS and SS selectors with values derived from bits 63:48 of the IA32_STAR MSR. However, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSRET instruction does not ensure this correspondence.

The SYSRET instruction does not modify the stack pointer (ESP or RSP). For that reason, it is necessary for software to switch to the user stack. The OS may load the user stack pointer (if it was saved after SYSCALL) before executing SYSRET; alternatively, user code may load the stack pointer (if it was saved before SYSCALL) after receiving control from SYSRET.

If the OS loads the stack pointer before executing SYSRET, it must ensure that the handler of any interrupt or exception delivered between restoring the stack pointer and successful execution of SYSRET is not invoked with the user stack. It can do so using approaches such as the following:

- External interrupts. The OS can prevent an external interrupt from being delivered by clearing EFLAGS.IF before loading the user stack pointer.
- Nonmaskable interrupts (NMIs). The OS can ensure that the NMI handler is invoked with the correct stack by using the interrupt stack table (IST) mechanism for gate 2 (NMI) in the IDT (see Section 7.14.5, "Interrupt Stack Table," in Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A).
- General-protection exceptions (#GP). The SYSRET instruction generates #GP(0) if the value of RCX is not canonical. The OS can address this possibility using one or more of the following approaches:
  - — Confirming that the value of RCX is canonical before executing SYSRET.
  - — Using paging to ensure that the SYSCALL instruction will never save a non-canonical value into RCX.
  - — Using the IST mechanism for gate 13 (#GP) in the IDT.

When shadow stacks are enabled at privilege level 3 the instruction loads SSP with value from IA32_PL3_SSP MSR. Refer to Chapter 6, "Procedure Calls, Interrupts, and Exceptions," and Chapter 18, "Control-flow Enforcement Technology (CET)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for additional CET details.

The instruction cannot be executed when FRED transitions are enabled. An operating system that has enabled FRED transitions should use ERETU instead.

1. Regardless of the value of R11, the RF and VM flags are always 0 in RFLAGS after execution of SYSRET. In addition, all reserved bits in RFLAGS retain the fixed values.

Instruction ordering. Instructions following a SYSRET may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the SYSRET have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

Operation

```
IF (CS.L ≠ 1 ) or (IA32_EFER.LMA ≠ 1) or (IA32_EFER.SCE ≠ 1) or (CR4.FRED = 1)
(* Not in 64-Bit Mode or SYSCALL/SYSRET not enabled in IA32_EFER or FRED enabled *)
    THEN #UD; FI;
IF (CPL ≠ 0) THEN #GP(0); FI;

IF (operand size is 64-bit)
    THEN (* Return to 64-Bit Mode *)
        IF (RCX is not canonical) THEN #GP(0);
        RIP := RCX;
    ELSE (* Return to Compatibility Mode *)
        RIP := ECX;
FI;
RFLAGS := (R11 & 3C7FD7H) | 2;          (* Clear RF, VM, reserved bits; set bit 1 *)

IF (operand size is 64-bit)
    THEN CS.Selector := IA32_STAR[63:48]+16;
    ELSE CS.Selector := IA32_STAR[63:48];
FI;
CS.Selector := CS.Selector OR 3;        (* RPL forced to 3 *)
(* Set rest of CS to a fixed value *)
CS.Base := 0;                           (* Flat segment *)
CS.Limit := FFFFFH;                     (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type := 11;                          (* Execute/read code, accessed *)
CS.S := 1;
CS.DPL := 3;
CS.P := 1;
IF (operand size is 64-bit)
    THEN (* Return to 64-Bit Mode *)
        CS.L := 1;                      (* 64-bit code segment *)
        CS.D := 0;                      (* Required if CS.L = 1 *)
    ELSE (* Return to Compatibility Mode *)
        CS.L := 0;                      (* Compatibility mode *)
        CS.D := 1;                      (* 32-bit code segment *)
FI;
CS.G := 1;                              (* 4-KByte granularity *)
CPL := 3;
IF ShadowStackEnabled(CPL)
    SSP := IA32_PL3_SSP;
FI;

SS.Selector := (IA32_STAR[63:48]+8) OR 3;   (* RPL forced to 3 *)
(* Set rest of SS to a fixed value *)
SS.Base := 0;                           (* Flat segment *)
SS.Limit := FFFFFH;                     (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type := 3;                           (* Read/write data, accessed *)
SS.S := 1;
SS.DPL := 3;
SS.P := 1;
SS.B := 1;                              (* 32-bit stack segment*)
SS.G := 1;                              (* 4-KByte granularity *)
```

Flags Affected

All.

Protected Mode Exceptions

#UD	The SYSRET instruction is not recognized in protected mode.

Real-Address Mode Exceptions

#UD	The SYSRET instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The SYSRET instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	The SYSRET instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD	If IA32_EFER.SCE = 0.
	If the LOCK prefix is used.
	If CR4.FRED = 1.
#GP(0)	If CPL ≠ 0.
	If the return is to 64-bit mode and RCX contains a non-canonical address.

TCMMIMFP16PS/TCMMRLFP16PS—Matrix Multiplication of Complex Tiles Accumulated into Packed Single Precision Tile

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 6C 11:rrr:bbb TCMMIMFP16PS tmm1, tmm2, tmm3	A	V/N.E.	AMX_COMPLEX	Matrix multiply complex elements from tmm2 and tmm3, and accumulate the imaginary part into single precision elements in tmm1.
VEX.128.NP.0F38.W0 6C 11:rrr:bbb TCMMRLFP16PS tmm1, tmm2, tmm3	A	V/N.E.	AMX_COMPLEX	Matrix multiply complex elements from tmm2 and tmm3, and accumulate the real part into single precision elements in tmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	VEX.vvvv (r)	N/A

Description

These instructions perform matrix multiplication of two tiles containing complex elements and accumulate the results into a packed single precision tile. Each dword element in input tiles tmm2 and tmm3 is interpreted as a complex number with FP16 real part and FP16 imaginary part.

TCMMRLFP16PS calculates the real part of the result. For each possible combination of (row of tmm2, column of tmm3), the instruction performs a set of multiplication and accumulations on all corresponding complex numbers (one from tmm2 and one from tmm3). The real part of the tmm2 element is multiplied with the real part of the corresponding tmm3 element, and the negated imaginary part of the tmm2 element is multiplied with the imaginary part of the corresponding tmm3 elements. The two accumulated results are added, and then accumulated into the corresponding row and column of tmm1.

TCMMIMFP16PS calculates the imaginary part of the result. For each possible combination of (row of tmm2, column of tmm3), the instruction performs a set of multiplication and accumulations on all corresponding complex numbers (one from tmm2 and one from tmm3). The imaginary part of the tmm2 element is multiplied with the real part of the corresponding tmm3 element, and the real part of the tmm2 element is multiplied with the imaginary part of the corresponding tmm3 elements. The two accumulated results are added, and then accumulated into the corresponding row and column of tmm1.

"Round to nearest even" rounding mode is used when doing each accumulation of the FMA. Output denormals are always flushed to zero but FP16 input denormals are not treated as zero.

MXCSR is not consulted nor updated.

Any attempt to execute these instructions inside an Intel TSX transaction will result in a transaction abort.

Operation

TCMMIMFP16PS tsrcdest, tsrc1, tsrc2

```
// C = m x n (tsrcdest), A = m x k (tsrc1), B = k x n (tsrc2)

# src1 and src2 elements are pairs of fp16
elements_src1 := tsrc1.colsb / 4
elements_dest := tsrcdest.colsb / 4
elements_temp := tsrcdest.colsb / 2  // Count is in fp16 prior to horizontal

for m in 0 ... tsrcdest.rows-1:
    temp1[ 0 ... elements_temp-1] := 0
    for k in 0 ... elements_src1-1:
        for n in 0 ... elements_dest-1:
```

```
                s1e = cvt_fp16_to_fp32(tsrc1.row[m].fp16[2*k+0])     // real
                s2e = cvt_fp16_to_fp32(tsrc2.row[k].fp16[2*n+0])     // real
                s1o = cvt_fp16_to_fp32(tsrc1.row[m].fp16[2*k+1])     // imaginary
                s2o = cvt_fp16_to_fp32(tsrc2.row[k].fp16[2*n+1])     // imaginary

                // FP32 FMA with DAZ=FTZ=1, RNE rounding.
                // MXCSR is neither consulted nor updated.
                // No exceptions raised or denoted.

                temp1.fp32[2*n+0] = fma32(temp1.fp32[2*n+0], s1o, s2e, daz=1, ftz=1, sae=1, rc=RNE)
                temp1.fp32[2*n+1] = fma32(temp1.fp32[2*n+1], s1e, s2o, daz=1, ftz=1, sae=1, rc=RNE)

    for n in 0 ... elements_dest-1:
        // DAZ=FTZ=1, RNE rounding.
        // MXCSR is neither consulted nor updated.
        // No exceptions raised or denoted.
        tmpf32 := temp1.fp32[2*n] + temp1.fp32[2*n+1]
        srcdest.row[m].fp32[n] := srcdest.row[m].fp32[n] + tmpf32
    write_row_and_zero(tsrcdest, m, tmp, tsrcdest.colsb)

zero_upper_rows(tsrcdest, tsrcdest.rows)
zero_tileconfig_start()
```

TCMMRLFP16PS tsrcdest, tsrc1, tsrc2

```
// C = m x n (tsrcdest), A = m x k (tsrc1), B = k x n (tsrc2)

# src1 and src2 elements are pairs of fp16
elements_src1 := tsrc1.colsb / 4
elements_dest := tsrcdest.colsb / 4
elements_temp := tsrcdest.colsb / 2 // Count is in fp16 prior to horizontal

for m in 0 ... tsrcdest.rows-1:
    temp1[ 0 ... elements_temp-1 ] := 0
    for k in 0 ... elements_src1-1:
        for n in 0 ... elements_dest-1:

            s1e = cvt_fp16_to_fp32(tsrc1.row[m].fp16[2*k+0])     // real
            s2e = cvt_fp16_to_fp32(tsrc2.row[k].fp16[2*n+0])     // real
            s1o = cvt_fp16_to_fp32(-tsrc1.row[m].fp16[2*k+1])    // imaginary: "-" is for imaginary*imaginary
            s2o = cvt_fp16_to_fp32(tsrc2.row[k].fp16[2*n+1])     // imaginary

            //   FP32 FMA with DAZ=FTZ=1, RNE rounding.
            //   MXCSR is neither consulted nor updated.
            //   No exceptions raised or denoted.

            temp1.fp32[2*n+0] = fma32(temp1.fp32[2*n+0], s1e, s2e, daz=1, ftz=1, sae=1, rc=RNE)      // real
            temp1.fp32[2*n+1] = fma32(temp1.fp32[2*n+1], s1o, s2o, daz=1, ftz=1, sae=1, rc=RNE)      // imaginary

    for n in 0 ... elements_dest-1:
        // DAZ=FTZ=1, RNE rounding.
        // MXCSR is neither consulted nor updated.
        // No exceptions raised or denoted.
        tmpf32 := temp1.fp32[2*n] + temp1.fp32[2*n+1]
```

```
            srcdest.row[m].fp32[n] := srcdest.row[m].fp32[n] + tmpf32
        write_row_and_zero(tsrcdest, m, tmp, tsrcdest.colsb)

zero_upper_rows(tsrcdest, tsrcdest.rows)
zero_tileconfig_start()
```

Flags Affected

None.

Exceptions

AMX-E4; see Section 2.10, "Intel® AMX Instruction Exception Classes," for details.

TDPBF16PS—Dot Product of BF16 Tiles Accumulated into Packed Single Precision Tile

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.F3.0F38.W0 5C 11:rrr:bbb TDPBF16PS tmm1, tmm2, tmm3	A	V/N.E.	AMX_BF16	Matrix multiply BF16 elements from tmm2 and tmm3, and accumulate the packed single precision elements in tmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	VEX.vvvv (r)	N/A

Description

This instruction performs a set of SIMD dot-products of two BF16 elements and accumulates the results into a packed single precision tile. Each dword element in input tiles tmm2 and tmm3 is interpreted as a BF16 pair. For each possible combination of (row of tmm2, column of tmm3), the instruction performs a set of SIMD dot-products on all corresponding BF16 pairs (one pair from tmm2 and one pair from tmm3), adds the results of those dot-products, and then accumulates the result into the corresponding row and column of tmm1.

"Round to nearest even" rounding mode is used when doing each accumulation of the FMA. Output denormals are always flushed to zero and input denormals are always treated as zero. MXCSR is not consulted nor updated.

Any attempt to execute the TDPBF16PS instruction inside a TSX transaction will result in a transaction abort.

Operation

```
define make_fp32(x):
    // The x parameter is bfloat16. Pack it in to upper 16b of a dword.
    // The bit pattern is a legal fp32 value. Return that bit pattern.
    dword: = 0
    dword[31:16] := x
return dword

TDPBF16PS tsrcdest, tsrc1, tsrc2
// C = m x n (tsrcdest), A = m x k (tsrc1), B = k x n (tsrc2)

# src1 and src2 elements are pairs of bfloat16
elements_src1 := tsrc1.colsb / 4
elements_src2 := tsrc2.colsb / 4
elements_dest := tsrcdest.colsb / 4
elements_temp := tsrcdest.colsb / 2        // Count is in bfloat16 prior to horizontal

for m in 0 … tsrcdest.rows-1:
    temp1[ 0 … elements_temp-1 ] := 0
    for k in 0 … elements_src1-1:
        for n in 0 … elements_dest-1:

            // FP32 FMA with DAZ=FTZ=1, RNE rounding.
            // MXCSR is neither consulted nor updated.
            // No exceptions raised or denoted.

            temp1.fp32[2*n+0] += make_fp32(tsrc1.row[m].bfloat16[2*k+0]) * make_fp32(tsrc2.row[k].bfloat16[2*n+0])
            temp1.fp32[2*n+1] += make_fp32(tsrc1.row[m].bfloat16[2*k+1]) * make_fp32(tsrc2.row[k].bfloat16[2*n+1])
```

```
    for n in 0 ... elements_dest-1:
        // DAZ=FTZ=1, RNE rounding.
        // MXCSR is neither consulted nor updated.
        // No exceptions raised or denoted.
        tmpf32 := temp1.fp32[2*n] + temp1.fp32[2*n+1]
        tsrcdest.row[m].fp32[n] := tsrcdest.row[m].fp32[n] + tmpf32
    write_row_and_zero(tsrcdest, m, tmp, tsrcdest.colsb)

zero_upper_rows(tsrcdest, tsrcdest.rows)
zero_tilecfg_start()
```

Intel C/C++ Compiler Intrinsic Equivalent

TDPBF16PS void _tile_dpbf16ps(__tile dst, __tile src1, __tile src2);

Flags Affected

None.

Exceptions

AMX-E4; see Section 2.10, "Intel® AMX Instruction Exception Classes," for details.

TDPBSSD/TDPBSUD/TDPBUSD/TDPBUUD—Dot Product of Signed/Unsigned Bytes with Dword Accumulation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.F2.0F38.W0 5E 11:rrr:bbb TDPBSSD tmm1, tmm2, tmm3	A	V/N.E.	AMX_INT8	Matrix multiply signed byte elements from tmm2 by signed byte elements from tmm3 and accumulate the dword elements in tmm1.
VEX.128.F3.0F38.W0 5E 11:rrr:bbb TDPBSUD tmm1, tmm2, tmm3	A	V/N.E.	AMX_INT8	Matrix multiply signed byte elements from tmm2 by unsigned byte elements from tmm3 and accumulate the dword elements in tmm1.
VEX.128.66.0F38.W0 5E 11:rrr:bbb TDPBUSD tmm1, tmm2, tmm3	A	V/N.E.	AMX_INT8	Matrix multiply unsigned byte elements from tmm2 by signed byte elements from tmm3 and accumulate the dword elements in tmm1.
VEX.128.NP.0F38.W0 5E 11:rrr:bbb TDPBUUD tmm1, tmm2, tmm3	A	V/N.E.	AMX_INT8	Matrix multiply unsigned byte elements from tmm2 by unsigned byte elements from tmm3 and accumulate the dword elements in tmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	VEX.vvvv (r)	N/A

Description

For each possible combination of (row of tmm2, column of tmm3), the instruction performs a set of SIMD dot-products on all corresponding four byte elements, one from tmm2 and one from tmm3, adds the results of those dot-products, and then accumulates the result into the corresponding row and column of tmm1. Each dword in input tiles tmm2 and tmm3 is interpreted as four byte elements. These may be signed or unsigned. Each letter in the two-letter pattern SU, US, SS, UU indicates the signed/unsigned nature of the values in tmm2 and tmm3, respectively.

Any attempt to execute the TDPBSSD/TDPBSUD/TDPBUSD/TDPBUUD instructions inside an Intel TSX transaction will result in a transaction abort.

Operation

```
define DPBD(c,x,y):// arguments are dwords
    if *x operand is signed*:
        extend_src1 := SIGN_EXTEND
    else:
        extend_src1 := ZERO_EXTEND

    if *y operand is signed*:
        extend_src2 := SIGN_EXTEND
    else:
        extend_src2 := ZERO_EXTEND

    p0dword := extend_src1(x.byte[0]) * extend_src2(y.byte[0])
    p1dword := extend_src1(x.byte[1]) * extend_src2(y.byte[1])
    p2dword := extend_src1(x.byte[2]) * extend_src2(y.byte[2])
    p3dword := extend_src1(x.byte[3]) * extend_src2(y.byte[3])

    c := c + p0dword + p1dword + p2dword + p3dword
```

TDPBSSD, TDPBSUD, TDPBUSD, TDPBUUD tsrcdest, tsrc1, tsrc2 (Register Only Version)

```
// C = m x n (tsrcdest), A = m x k (tsrc1), B = k x n (tsrc2)

tsrc1_elements_per_row := tsrc1.colsb / 4
tsrc2_elements_per_row := tsrc2.colsb / 4
tsrcdest_elements_per_row := tsrcdest.colsb / 4

for m in 0 ... tsrcdest.rows-1:
    tmp := tsrcdest.row[m]
    for k in 0 ... tsrc1_elements_per_row-1:
        for n in 0 ... tsrcdest_elements_per_row-1:
            DPBD( tmp.dword[n], tsrc1.row[m].dword[k], tsrc2.row[k].dword[n] )
    write_row_and_zero(tsrcdest, m, tmp, tsrcdest.colsb)

zero_upper_rows(tsrcdest, tsrcdest.rows)
zero_tilecfg_start()
```

Intel C/C++ Compiler Intrinsic Equivalent

TDPBSSD void _tile_dpbssd(__tile dst, __tile src1, __tile src2);
TDPBSUD void _tile_dpbsud(__tile dst, __tile src1, __tile src2);
TDPBUSD void _tile_dpbusd(__tile dst, __tile src1, __tile src2);
TDPBUUD void _tile_dpbuud(__tile dst, __tile src1, __tile src2);

Flags Affected

None.

Exceptions

AMX-E4; see Section 2.10, "Intel® AMX Instruction Exception Classes," for details.

TDPFP16PS—Dot Product of FP16 Tiles Accumulated into Packed Single Precision Tile

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.F2.0F38.W0 5C 11:rrr:bbb TDPFP16PS tmm1, tmm2, tmm3	A	V/N.E.	AMX_FP16	Matrix multiply FP16 elements from tmm2 and tmm3, and accumulate the packed single precision elements in tmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	VEX.vvvv (r)	N/A

Description

This instruction performs a set of SIMD dot-products of two FP16 elements and accumulates the results into a packed single precision tile. Each dword element in input tiles tmm2 and tmm3 is interpreted as a FP16 pair. For each possible combination of (row of tmm2, column of tmm3), the instruction performs a set of SIMD dot-products on all corresponding FP16 pairs (one pair from tmm2 and one pair from tmm3), adds the results of those dot-products, and then accumulates the result into the corresponding row and column of tmm1.

"Round to nearest even" rounding mode is used when doing each accumulation of the Fused Multiply-Add (FMA). Output FP32 denormals are always flushed to zero. Input FP16 denormals are always handled and not treated as zero.

MXCSR is not consulted nor updated.

Any attempt to execute the TDPFP16PS instruction inside an Intel TSX transaction will result in a transaction abort.

Operation

TDPFP16PS tsrcdest, tsrc1, tsrc2

```
// C = m x n (tsrcdest), A = m x k (tsrc1), B = k x n (tsrc2)

# src1 and src2 elements are pairs of fp16
elements_src1 := tsrc1.colsb / 4
elements_src2 := tsrc2.colsb / 4
elements_dest := tsrcdest.colsb / 4
elements_temp := tsrcdest.colsb / 2 // Count is in fp16 prior to horizontal

for m in 0 ... tsrcdest.rows-1:
    temp1[ 0 ... elements_temp-1 ] := 0
    for k in 0 ... elements_src1-1:
        for n in 0 ... elements_dest-1:

            // For this operation:
            // Handle FP16 denorms. Not forcing input FP16 denorms to 0.
            // FP32 FMA with DAZ=FTZ=1, RNE rounding.
            // MXCSR is neither consulted nor updated.
            // No exceptions raised or denoted.

            temp1.fp32[2*n+0] += cvt_fp16_to_fp32(tsrc1.row[m].fp16[2*k+0]) *cvt_fp16_to_fp32(tsrc2.row[k].fp16[2*n+0])
            temp1.fp32[2*n+1] += cvt_fp16_to_fp32(tsrc1.row[m].fp16[2*k+1]) *cvt_fp16_to_fp32(tsrc2.row[k].fp16[2*n+1])

    for n in 0 ... elements_dest-1:
        // DAZ=FTZ=1, RNE rounding.
        // MXCSR is neither consulted nor updated.
```

```
            // No exceptions raised or denoted.
            tmpf32 := temp1.fp32[2*n] + temp1.fp32[2*n+1]
            srcdest.row[m].fp32[n] := srcdest.row[m].fp32[n] + tmpf32
        write_row_and_zero(tsrcdest, m, tmp, tsrcdest.colsb)
zero_upper_rows(tsrcdest, tsrcdest.rows)
zero_tileconfig_start()
```

Flags Affected

None.

Exceptions

AMX-E4; see Section 2.10, "Intel® AMX Instruction Exception Classes," for details.

TEST—Logical Compare

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
A8 ib	TEST AL, imm8	I	Valid	Valid	AND imm8 with AL; set SF, ZF, PF according to result.
A9 iw	TEST AX, imm16	I	Valid	Valid	AND imm16 with AX; set SF, ZF, PF according to result.
A9 id	TEST EAX, imm32	I	Valid	Valid	AND imm32 with EAX; set SF, ZF, PF according to result.
REX.W + A9 id	TEST RAX, imm32	I	Valid	N.E.	AND imm32 sign-extended to 64-bits with RAX; set SF, ZF, PF according to result.
F6 /0 ib	TEST r/m8[1], imm8	MI	Valid	Valid	AND imm8 with r/m8; set SF, ZF, PF according to result.
F7 /0 iw	TEST r/m16, imm16	MI	Valid	Valid	AND imm16 with r/m16; set SF, ZF, PF according to result.
F7 /0 id	TEST r/m32, imm32	MI	Valid	Valid	AND imm32 with r/m32; set SF, ZF, PF according to result.
REX.W + F7 /0 id	TEST r/m64, imm32	MI	Valid	N.E.	AND imm32 sign-extended to 64-bits with r/m64; set SF, ZF, PF according to result.
84 /r	TEST r/m8[1], r8[1]	MR	Valid	Valid	AND r8 with r/m8; set SF, ZF, PF according to result.
85 /r	TEST r/m16, r16	MR	Valid	Valid	AND r16 with r/m16; set SF, ZF, PF according to result.
85 /r	TEST r/m32, r32	MR	Valid	Valid	AND r32 with r/m32; set SF, ZF, PF according to result.
REX.W + 85 /r	TEST r/m64, r64	MR	Valid	N.E.	AND r64 with r/m64; set SF, ZF, PF according to result.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	imm8/16/32	N/A	N/A
MI	ModRM:r/m (r)	imm8/16/32	N/A	N/A
MR	ModRM:r/m (r)	ModRM:reg (r)	N/A	N/A

Description

Computes the bit-wise logical AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result. The result is then discarded.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
TEMP := SRC1 AND SRC2;
SF := MSB(TEMP);

IF TEMP = 0
    THEN ZF := 1;
    ELSE ZF := 0;
FI:

PF := BitwiseXNOR(TEMP[0:7]);
CF := 0;
OF := 0;
```

(* AF is undefined *)

Flags Affected

The OF and CF flags are set to 0. The SF, ZF, and PF flags are set according to the result (see the "Operation" section above). The state of the AF flag is undefined.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

TESTUI—Determine User Interrupt Flag

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 01 ED TESTUI	ZO	V/I	UINTR	Copies the current value of UIF into EFLAGS.CF.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A	N/A

TESTUI copies the current value of the user interrupt flag (UIF) into EFLAGS.CF. This instruction can be executed regardless of CPL.

TESTUI may be executed normally inside a transactional region.

Operation

```
CF := UIF;
ZF := AF := OF := PF := SF := 0;
```

Flags Affected

The ZF, OF, AF, PF, SF flags are cleared and the CF flags to the value of the user interrupt flag.

Protected Mode Exceptions

#UD	The TESTUI instruction is not recognized in protected mode.

Real-Address Mode Exceptions

#UD	The TESTUI instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The TESTUI instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	The TESTUI instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used. If executed inside an enclave. If CR4.UINTR = 0. If CPUID.07H.00H:EDX.UINTR[5] = 0.

TILELOADD/TILELOADDT1—Load Tile

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.F2.0F38.W0 4B !(11):rrr:100 TILELOADD tmm1, sibmem	A	V/N.E.	AMX_TILE	Load data into tmm1 as specified by information in sibmem.
VEX.128.66.0F38.W0 4B !(11):rrr:100 TILELOADDT1 tmm1, sibmem	A	V/N.E.	AMX_TILE	Load data into tmm1 as specified by information in sibmem with hint to optimize data caching.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction is required to use SIB addressing. The index register serves as a stride indicator. If the SIB encoding omits an index register, the value zero is assumed for the content of the index register.

This instruction loads a tile destination with rows and columns as specified by the tile configuration. The "T1" version provides a hint to the implementation that the data would be reused but does not need to be resident in the nearest cache levels.

The TILECFG.start_row in the TILECFG data should be initialized to '0' in order to load the entire tile and is set to zero on successful completion of the TILELOADD instruction. TILELOADD is a restartable instruction and the TILECFG.start_row will be non-zero when restartable events occur during the instruction execution.

Only memory operands are supported and they can only be accessed using a SIB addressing mode, similar to the V[P]GATHER*/V[P]SCATTER* instructions.

Any attempt to execute the TILELOADD/TILELOADDT1 instructions inside an Intel TSX transaction will result in a transaction abort.

Operation

```
TILELOADD[,T1] tdest, tsib

start := tilecfg.start_row

zero_upper_rows(tdest,start)

membegin := tsib.base + displacement
// if no index register in the SIB encoding, the value zero is used.
stride := tsib.index << tsib.scale
nbytes := tdest.colsb
while start < tdest.rows:
    memptr := membegin + start * stride
    write_row_and_zero(tdest, start, read_memory(memptr, nbytes), nbytes)
    start := start + 1
zero_tilecfg_start()
// In the case of a memory fault in the middle of an instruction, the tilecfg.start_row := start
```

Intel C/C++ Compiler Intrinsic Equivalent

TILELOADD void _tile_loadd(__tile dst, const void *base, int stride);
TILELOADDT1 void _tile_stream_loadd(__tile dst, const void *base, int stride);

Flags Affected

None.

Exceptions

AMX-E3; see Section 2.10, “Intel® AMX Instruction Exception Classes,” for details.

TILERELEASE—Release Tile

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.NP.0F38.W0 49 C0 TILERELEASE	A	V/N.E.	AMX_TILE	Initialize TILECFG and TILEDATA.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	N/A	N/A	N/A	N/A

Description

This instruction returns TILECFG and TILEDATA to the INIT state.

Any attempt to execute the TILERELEASE instruction inside an Intel TSX transaction will result in a transaction abort.

Operation

```
zero_all_tile_data()
tilecfg := 0// equivalent to 64B of zeros
TILES_CONFIGURED := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

TILERELEASE void _tile_release(void);

Flags Affected

None.

Exceptions

AMX-E6; see Section 2.10, "Intel® AMX Instruction Exception Classes," for details.

TILESTORED—Store Tile

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.F3.0F38.W0 4B !(11):rrr:100 TILESTORED sibmem, tmm1	A	V/N.E.	AMX_TILE	Store a tile in sibmem as specified in tmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

This instruction is required to use SIB addressing. The index register serves as a stride indicator. If the SIB encoding omits an index register, the value zero is assumed for the content of the index register.

This instruction stores a tile source of rows and columns as specified by the tile configuration.

The TILECFG.start_row in the TILECFG data should be initialized to '0' in order to store the entire tile and are set to zero on successful completion of the TILESTORED instruction. TILESTORED is a restartable instruction and the TILECFG.start_row will be non-zero when restartable events occur during the instruction execution.

Only memory operands are supported and they can only be accessed using a SIB addressing mode, similar to the V[P]GATHER*/V[P]SCATTER* instructions.

Any attempt to execute the TILESTORED instruction inside an Intel TSX transaction will result in a transaction abort.

Operation

```
TILESTORED tsib, tsrc

start := tilecfg.start_row

membegin := tsib.base + displacement
// if no index register in the SIB encoding, the value zero is used.
stride := tsib.index << tsib.scale

while start < tdest.rows:
    memptr := membegin + start * stride
    write_memory(memptr, tsrc.colsb, tsrc.row[start])
    start := start + 1
zero_tilecfg_start()
// In the case of a memory fault in the middle of an instruction, the tilecfg.start_row := start
```

Intel C/C++ Compiler Intrinsic Equivalent

TILESTORED void _tile_stored(__tile src, void *base, int stride);

Flags Affected

None.

Exceptions

AMX-E3; see Section 2.10, "Intel® AMX Instruction Exception Classes," for details.

TILEZERO—Zero Tile

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.F2.0F38.W0 49 11:rrr:000 TILEZERO tmm1	A	V/N.E.	AMX_TILE	Zero the destination tile.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	N/A	N/A	N/A

Description

This instruction zeroes the destination tile.

Any attempt to execute the TILEZERO instruction inside an Intel TSX transaction will result in a transaction abort.

Operation

```
TILEZERO tdest

nbytes := palette_table[palette_id].bytes_per_row

for i in 0 ... palette_table[palette_id].max_rows-1:
    for j in 0 ... nbytes-1:
        tdest.row[i].byte[j] := 0
zero_tilecfg_start()
```

Intel C/C++ Compiler Intrinsic Equivalent

TILEZERO void _tile_zero(__tile dst);

Flags Affected

None.

Exceptions

AMX-E5; see Section 2.10, "Intel® AMX Instruction Exception Classes," for details.

TPAUSE—Timed PAUSE

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F AE /6 TPAUSE r32, <edx>, <eax>	A	V/V	WAITPKG	Directs the processor to enter an implementation-dependent optimized state until the TSC reaches the value in EDX:EAX.

Instruction Operand Encoding[1]

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (r)	N/A	N/A	N/A

Description

TPAUSE instructs the processor to enter an implementation-dependent optimized state. There are two such optimized states to choose from: light-weight power/performance optimized state, and improved power/performance optimized state. The selection between the two is governed by the explicit input register bit[0] source operand.

TPAUSE is available when CPUID.07H.00H:ECX.WAITPKG[5] is enumerated as 1. TPAUSE may be executed at any privilege level. This instruction's operation is the same in non-64-bit modes and in 64-bit mode.

Unlike PAUSE, the TPAUSE instruction will not cause an abort when used inside a transactional region, described in the chapter Chapter 16, "Programming with Intel® AVX10," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

The input register contains information such as the preferred optimized state the processor should enter as described in the following table. Bits other than bit 0 are reserved and will result in #GP if non-zero.

Table 4-23. TPAUSE Input Register Bit Definitions

Bit Value	State Name	Wakeup Time	Power Savings	Other Benefits
bit[0] = 0	C0.2	Slower	Larger	Improves performance of the other SMT thread(s) on the same core.
bit[0] = 1	C0.1	Faster	Smaller	N/A
bits[31:1]	N/A	N/A	N/A	Reserved

The instruction execution wakes up when the time-stamp counter reaches or exceeds the implicit EDX:EAX 64-bit input value.

Prior to executing the TPAUSE instruction, an operating system may specify the maximum delay it allows the processor to suspend its operation. It can do so by writing TSC-quanta value to the following 32-bit MSR (IA32_UMWAIT_CONTROL at MSR index E1H):

- IA32_UMWAIT_CONTROL[31:2] — Determines the maximum time in TSC-quanta that the processor can reside in either C0.1 or C0.2. A zero value indicates no maximum time. The maximum time value is a 32-bit value where the upper 30 bits come from this field and the lower two bits are zero.
- IA32_UMWAIT_CONTROL[1] — Reserved.
- IA32_UMWAIT_CONTROL[0] — C0.2 is not allowed by the OS. Value of "1" means all C0.2 requests revert to C0.1.

If the processor that executed a TPAUSE instruction wakes due to the expiration of the operating system time-limit, the instructions sets RFLAGS.CF; otherwise, that flag is cleared.

The following additional events cause the processor to exit the implementation-dependent optimized state: a store to the read-set range within the transactional region, an NMI or SMI, a debug exception, a machine check exception, the BINIT# signal, the INIT# signal, and the RESET# signal.

1. The Mod field of the ModR/M byte must have value 11B.

Other implementation-dependent events may cause the processor to exit the implementation-dependent optimized state proceeding to the instruction following TPAUSE. In addition, an external interrupt causes the processor to exit the implementation-dependent optimized state regardless of whether maskable-interrupts are inhibited (EFLAGS.IF =0). It should be noted that if maskable-interrupts are inhibited execution will proceed to the instruction following TPAUSE.

Operation

```
os_deadline := TSC+(IA32_UMWAIT_CONTROL[31:2]<<2)
instr_deadline := UINT64(EDX:EAX)

IF os_deadline < instr_deadline:
    deadline := os_deadline
    using_os_deadline := 1
ELSE:
    deadline := instr_deadline
    using_os_deadline := 0

WHILE TSC < deadline:
    implementation_dependent_optimized_state(Source register, deadline, IA32_UMWAIT_CONTROL[0])

IF using_os_deadline AND TSC ≥ deadline:
    RFLAGS.CF := 1
ELSE:
    RFLAGS.CF := 0

RFLAGS.AF,PF,SF,ZF,OF := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

TPAUSE uint8_t _tpause(uint32_t control, uint64_t counter);

Numeric Exceptions

None.

Exceptions (All Operating Modes)

#GP(0)	If src[31:1] != 0.
	If CR4.TSD = 1 and CPL != 0.
#UD	If CPUID.07H.00H:ECX.WAITPKG[5]=0.

TZCNT—Count the Number of Trailing Zero Bits

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F BC /r TZCNT r16, r/m16	A	V/V	BMI1	Count the number of trailing zero bits in r/m16, return result in r16.
F3 0F BC /r TZCNT r32, r/m32	A	V/V	BMI1	Count the number of trailing zero bits in r/m32, return result in r32.
F3 REX.W 0F BC /r TZCNT r64, r/m64	A	V/N.E.	BMI1	Count the number of trailing zero bits in r/m64, return result in r64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

TZCNT counts the number of trailing least significant zero bits in source operand (second operand) and returns the result in the destination operand (first operand). TZCNT is an extension of the BSF instruction. The key difference between the TZCNT and BSF instructions is that when the source operand is zero, TZCNT outputs the operand size to the destination operand, whereas BSF leaves the destination operand unmodified.

On processors that do not support TZCNT, the instruction byte encoding is executed as BSF.

Operation

```
temp := 0
DEST := 0
DO WHILE ( (temp < OperandSize) and (SRC[ temp] = 0) )

    temp := temp +1
    DEST := DEST+ 1
OD

IF DEST = OperandSize
    CF := 1
ELSE
    CF := 0
FI

IF DEST = 0
    ZF := 1
ELSE
    ZF := 0
FI
```

Flags Affected

ZF is set to 1 in case of zero output (least significant bit of the source is set), and to 0 otherwise, CF is set to 1 if the input was zero and cleared otherwise. OF, SF, PF, and AF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

TZCNT unsigned __int32 _tzcnt_u32(unsigned __int32 src);
TZCNT unsigned __int64 _tzcnt_u64(unsigned __int64 src);

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If LOCK prefix is used.

Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	For an illegal address in the SS segment.
#UD	If LOCK prefix is used.

Virtual 8086 Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If LOCK prefix is used.

UCOMISD—Unordered Compare Scalar Double Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2E /r UCOMISD xmm1, xmm2/m64	A	V/V	SSE2	Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
VEX.LIG.66.0F.WIG 2E /r VUCOMISD xmm1, xmm2/m64	A	V/V	AVX	Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
EVEX.LLIG.66.0F.W1 2E /r VUCOMISD xmm1, xmm2/m64{sae}	B	V/V	AVX512F OR AVX10.1	Compare low double precision floating-point values in xmm1 and xmm2/m64 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r)	ModRM:r/m (r)	N/A	N/A
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Performs an unordered compare of the double precision floating-point values in the low quadwords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF, and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 64 bit memory

location.

The UCOMISD instruction differs from the COMISD instruction in that it signals a SIMD floating-point invalid operation exception (#I) only when a source operand is an SNaN. The COMISD instruction signals an invalid operation exception only if a source operand is either an SNaN or a QNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISD is encoded with VEX.L=0. Encoding VCOMISD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

(V)UCOMISD (All Versions)

```
RESULT := UnorderedCompare(DEST[63:0] <> SRC[63:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
    UNORDERED: ZF,PF,CF := 111;
    GREATER_THAN: ZF,PF,CF := 000;
    LESS_THAN: ZF,PF,CF := 001;
    EQUAL: ZF,PF,CF := 100;
ESAC;
OF, AF, SF := 0; }
```

Intel C/C++ Compiler Intrinsic Equivalent

VUCOMISD int _mm_comi_round_sd(__m128d a, __m128d b, int imm, int sae);
UCOMISD int _mm_ucomieq_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomilt_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomile_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomigt_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomige_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomineq_sd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid (if SNaN operands), Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions," additionally:

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Table 2-50, "Type E3NF Class Exception Conditions."

UCOMISS—Unordered Compare Scalar Single Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 2E /r UCOMISS xmm1, xmm2/m32	A	V/V	SSE	Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
VEX.LIG.0F.WIG 2E /r VUCOMISS xmm1, xmm2/m32	A	V/V	AVX	Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
EVEX.LLIG.0F.W0 2E /r VUCOMISS xmm1, xmm2/m32{sae}	B	V/V	AVX512F OR AVX10.1	Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r)	ModRM:r/m (r)	N/A	N/A
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Compares the single precision floating-point values in the low doublewords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF, and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 32 bit memory location.

The UCOMISS instruction differs from the COMISS instruction in that it signals a SIMD floating-point invalid operation exception (#I) only if a source operand is an SNaN. The COMISS instruction signals an invalid operation exception when a source operand is either a QNaN or SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISS is encoded with VEX.L=0. Encoding VCOMISS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

(V)UCOMISS (All Versions)

```
RESULT := UnorderedCompare(DEST[31:0] <> SRC[31:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
    UNORDERED: ZF,PF,CF := 111;
    GREATER_THAN: ZF,PF,CF := 000;
    LESS_THAN: ZF,PF,CF := 001;
    EQUAL: ZF,PF,CF := 100;
ESAC;
OF, AF, SF := 0; }
```

Intel C/C++ Compiler Intrinsic Equivalent

VUCOMISS int _mm_comi_round_ss(__m128 a, __m128 b, int imm, int sae);
UCOMISS int _mm_ucomieq_ss(__m128 a, __m128 b);
UCOMISS int _mm_ucomilt_ss(__m128 a, __m128 b);
UCOMISS int _mm_ucomile_ss(__m128 a, __m128 b);

UCOMISS int _mm_ucomigt_ss(__m128 a, __m128 b);
UCOMISS int _mm_ucomige_ss(__m128 a, __m128 b);
UCOMISS int _mm_ucomineq_ss(__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Invalid (if SNaN Operands), Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions," additionally:

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Table 2-50, "Type E3NF Class Exception Conditions."

UD—Undefined Instruction

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F FF /r	UD0[1] r32, r/m32	RM	Valid	Valid	Raise invalid opcode exception.
0F B9 /r	UD1 r32, r/m32	RM	Valid	Valid	Raise invalid opcode exception.
0F 0B	UD2	ZO	Valid	Valid	Raise invalid opcode exception.
D6	UDB	ZO	Valid	N.E.[2]	Raise invalid opcode exception.

NOTES:

1. Some processors decode the UD0 instruction without a ModR/M byte. As a result, those processors would deliver an invalid-opcode exception instead of a fault on instruction fetch when the instruction with a ModR/M byte (and any implied bytes) would cross a page or segment boundary.
2. Outside 64-bit mode, D6H encodes SALC.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A
RM	ModRM:reg (r)	ModRM:r/m (r)	N/A	N/A

Description

Generates an invalid opcode exception. This instruction is provided for software testing to explicitly generate an invalid opcode exception. The opcodes for this instruction are reserved for this purpose.

Other than raising the invalid opcode exception, this instruction has no effect on processor state or memory.

Even though it is the execution of the UD instruction that causes the invalid opcode exception, the instruction pointer saved by delivery of the exception references the UD instruction (and not the following instruction).

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
#UD (* Generates invalid opcode exception *);
```

Flags Affected

None.

Exceptions (All Operating Modes)

#UD	Raises an invalid opcode exception in all operating modes. (UDB does so only in 64-bit mode.)

UIRET—User-Interrupt Return

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 01 EC UIRET	ZO	V/I	UINTR	Return from handling a user interrupt.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A	N/A

Description

UIRET returns from the handling of a user interrupt. It can be executed regardless of CPL.

Execution of UIRET inside a transactional region causes a transactional abort; the abort loads EAX as it would have had it been due to an execution of IRET.

UIRET can be tracked by Architectural Last Branch Records (LBRs), Intel Processor Trace (Intel PT), and Performance Monitoring. For both Intel PT and LBRs, UIRET is recorded in precisely the same manner as IRET. Hence for LBRs, UIRETs fall into the OTHER_BRANCH category, which implies that IA32_LBR_CTL.OTHER_BRANCH[bit 22] must be set to record user-interrupt delivery, and that the IA32_LBR_x_INFO.BR_TYPE field will indicate OTHER_BRANCH for any recorded user interrupt. For Intel PT, control flow tracing must be enabled by setting IA32_RTIT_CTL.BranchEn[bit 13].

UIRET will also increment performance counters for which counting BR_INST_RETIRED.FAR_BRANCH is enabled.

Operation

```
Pop tempRIP;
Pop tempRFLAGS; // see below for how this is used to load RFLAGS
Pop tempRSP;
IF tempRIP is not canonical in current paging mode
    THEN #GP(0);
FI;
IF ShadowStackEnabled(CPL)
    THEN
        PopShadowStack SSRIP;
        IF SSRIP ≠ tempRIP
            THEN #CP (FAR-RET/IRET);
        FI;
FI;
RIP := tempRIP;
// update in RFLAGS only CF, PF, AF, ZF, SF, TF, DF, OF, NT, RF, AC, and ID
RFLAGS := (RFLAGS & ~254DD5H) | (tempRFLAGS & 254DD5H);
RSP := tempRSP;
IF CPUID.07H.01H:EDX.UIRET_UIF[17] = 1
    THEN UIF := tempRFLAGS[1];
    ELSE UIF := 1;
FI;
Clear any cache-line monitoring established by MONITOR or UMONITOR;
```

Flags Affected

See the Operation section.

Protected Mode Exceptions

#UD	The UIRET instruction is not recognized in protected mode.

Real-Address Mode Exceptions

#UD	The UIRET instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The UIRET instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	The UIRET instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#GP(0)	If the return instruction pointer is non-canonical.
#SS(0)	If an attempt to pop a value off the stack causes a non-canonical address to be referenced.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#CP	If return instruction pointer from stack and shadow stack do not match.
#UD	If the LOCK prefix is used.
	If executed inside an enclave.
	If CR4.UINTR = 0.
	If CPUID.07H.00H:EDX.UINTR[5] = 0.

UMONITOR—User Level Set Up Monitor Address

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F AE /6 UMONITOR r16/r32/r64	A	V/V	WAITPKG	Sets up a linear address range to be monitored by hardware and activates the monitor. The address range should be a write-back memory caching type. The address is contained in r16/r32/r64.

Instruction Operand Encoding[1]

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (r)	N/A	N/A	N/A

Description

The UMONITOR instruction arms address monitoring hardware using an address specified in the source register (the address range that the monitoring hardware checks for store operations can be determined by using the CPUID.05H). A store to an address within the specified address range triggers the monitoring hardware. The state of monitor hardware is used by UMWAIT.

The content of the source register is an effective address. By default, the DS segment is used to create a linear address that is monitored. Segment overrides can be used. The address range must use memory of the write-back type. Only write-back memory is guaranteed to correctly trigger the monitoring hardware. Additional information on determining what address range to use in order to prevent false wake-ups is described in Chapter 11, "Multiple-Processor Management," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

The UMONITOR instruction is ordered as a load operation with respect to other memory transactions. The instruction is subject to the permission checking and faults associated with a byte load. Like a load, UMONITOR sets the A-bit but not the D-bit in page tables.

UMONITOR and UMWAIT are available when CPUID.07H.00H:ECX.WAITPKG[5] is enumerated as 1. UMONITOR and UMWAIT may be executed at any privilege level. Except for the width of the source register, the instruction's operation is the same in non-64-bit modes and in 64-bit mode.

UMONITOR does not interoperate with the legacy MWAIT instruction. If UMONITOR was executed prior to executing MWAIT and following the most recent execution of the legacy MONITOR instruction, MWAIT will not enter an optimized state. Execution will continue to the instruction following MWAIT.

The UMONITOR instruction causes a transactional abort when used inside a transactional region.

The width of the source register (16b, 32b or 64b) is determined by the effective addressing width, which is affected in the standard way by the machine mode settings and 67 prefix.

Operation

UMONITOR sets up an address range for the monitor hardware using the content of source register as an effective address and puts the monitor hardware in armed state. A store to the specified address range will trigger the monitor hardware.

Intel C/C++ Compiler Intrinsic Equivalent

UMONITOR void _umonitor(void *address);

Numeric Exceptions

None.

1. The Mod field of the ModR/M byte must have value 11B.

Protected Mode Exceptions

#GP(0)	If the specified segment is not SS and the source register is outside the specified segment limit.
	If the specified segment register contains a NULL segment selector.
#SS(0)	If the specified segment is SS and the source register is outside the SS segment limit.
#PF(fault-code)	For a page fault.
#UD	If CPUID.07H.00H:ECX.WAITPKG[5]=0.
	If the LOCK prefix is used.

Real Address Mode Exceptions

#GP	If the specified segment is not SS and the source register is outside of the effective address space from 0 to FFFFH.
#SS	If the specified segment is SS and the source register is outside of the effective address space from 0 to FFFFH.
#UD	If CPUID.07H.00H:ECX.WAITPKG[5] =0.
	If the LOCK prefix is used.

Virtual 8086 Mode Exceptions

Same exceptions as in real address mode; additionally:

#PF(fault-code)	For a page fault.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the specified segment is not SS and the linear address is in non-canonical form.
#SS(0)	If the specified segment is SS and the source register is in non-canonical form.
#PF(fault-code)	For a page fault.
#UD	If CPUID.07H.00H:ECX.WAITPKG[5] =0.
	If the LOCK prefix is used.

UMWAIT—User Level Monitor Wait

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F AE /6 UMWAIT r32, <edx>, <eax>	A	V/V	WAITPKG	A hint that allows the processor to stop instruction execution and enter an implementation-dependent optimized state until occurrence of a class of events.

Instruction Operand Encoding[1]

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (r)	N/A	N/A	N/A

Description

UMWAIT instructs the processor to enter an implementation-dependent optimized state while monitoring a range of addresses. The optimized state may be either a light-weight power/performance optimized state or an improved power/performance optimized state. The selection between the two states is governed by the explicit input register bit[0] source operand.

UMWAIT is available when CPUID.07H.00H:ECX.WAITPKG[5] is enumerated as 1. UMWAIT may be executed at any privilege level. This instruction's operation is the same in non-64-bit modes and in 64-bit mode.

The input register contains information such as the preferred optimized state the processor should enter as described in the following table. Bits other than bit 0 are reserved and will result in #GP if nonzero.

Table 4-24. UMWAIT Input Register Bit Definitions

Bit Value	State Name	Wakeup Time	Power Savings	Other Benefits
bit[0] = 0	C0.2	Slower	Larger	Improves performance of the other SMT thread(s) on the same core.
bit[0] = 1	C0.1	Faster	Smaller	N/A
bits[31:1]	N/A	N/A	N/A	Reserved

The instruction wakes up when the time-stamp counter reaches or exceeds the implicit EDX:EAX 64-bit input value (if the monitoring hardware did not trigger beforehand).

Prior to executing the UMWAIT instruction, an operating system may specify the maximum delay it allows the processor to suspend its operation. It can do so by writing TSC-quanta value to the following 32bit MSR (IA32_UMWAIT_CONTROL at MSR index E1H):

- IA32_UMWAIT_CONTROL[31:2] — Determines the maximum time in TSC-quanta that the processor can reside in either C0.1 or C0.2. A zero value indicates no maximum time. The maximum time value is a 32-bit value where the upper 30 bits come from this field and the lower two bits are zero.
- IA32_UMWAIT_CONTROL[1] — Reserved.
- IA32_UMWAIT_CONTROL[0] — C0.2 is not allowed by the OS. Value of "1" means all C0.2 requests revert to C0.1.

If the processor that executed a UMWAIT instruction wakes due to the expiration of the operating system time-limit, the instructions sets RFLAGS.CF; otherwise, that flag is cleared.

The UMWAIT instruction causes a transactional abort when used inside a transactional region.

The UMWAIT instruction operates with the UMONITOR instruction. The two instructions allow the definition of an address at which to wait (UMONITOR) and an implementation-dependent optimized operation to perform while waiting (UMWAIT). The execution of UMWAIT is a hint to the processor that it can enter an implementation-dependent-optimized state while waiting for an event or a store operation to the address range armed by UMONITOR. The UMWAIT instruction will not wait (will not enter an implementation-dependent optimized state) if any of the

1. The Mod field of the ModR/M byte must have value 11B.

following instructions were executed before UMWAIT and after the most recent execution of UMONITOR: IRET, MONITOR, SYSEXIT, SYSRET, and far RET (the last if it is changing CPL).

The following additional events cause the processor to exit the implementation-dependent optimized state: a store to the address range armed by the UMONITOR instruction, an NMI or SMI, a debug exception, a machine check exception, the BINIT# signal, the INIT# signal, and the RESET# signal. Other implementation-dependent events may also cause the processor to exit the implementation-dependent optimized state.

In addition, an external interrupt causes the processor to exit the implementation-dependent optimized state regardless of whether maskable-interrupts are inhibited (EFLAGS.IF =0).

Following exit from the implementation-dependent-optimized state, control passes to the instruction after the UMWAIT instruction. A pending interrupt that is not masked (including an NMI or an SMI) may be delivered before execution of that instruction.

Unlike the HLT instruction, the UMWAIT instruction does not restart at the UMWAIT instruction following the handling of an SMI.

If the preceding UMONITOR instruction did not successfully arm an address range or if UMONITOR was not executed prior to executing UMWAIT and following the most recent execution of the legacy MONITOR instruction (UMWAIT does not interoperate with MONITOR), then the processor will not enter an optimized state. Execution will continue to the instruction following UMWAIT.

A store to the address range armed by the UMONITOR instruction will cause the processor to exit UMWAIT if either the store was originated by other processor agents or the store was originated by a non-processor agent.

Operation

```
os_deadline := TSC+(IA32_UMWAIT_CONTROL[31:2]<<2)
instr_deadline := UINT64(EDX:EAX)

IF os_deadline < instr_deadline:
    deadline := os_deadline
    using_os_deadline := 1
ELSE:
    deadline := instr_deadline
    using_os_deadline := 0

WHILE monitor hardware armed AND TSC < deadline:
    implementation_dependent_optimized_state(Source register, deadline, IA32_UMWAIT_CONTROL[0] )

IF using_os_deadline AND TSC ≥ deadline:
    RFLAGS.CF := 1
ELSE:
    RFLAGS.CF := 0

RFLAGS.AF,PF,SF,ZF,OF := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

UMWAIT uint8_t _umwait(uint32_t control, uint64_t counter);

Numeric Exceptions

None.

Exceptions (All Operating Modes)

#GP(0)	If src[31:1] != 0.
	If CR4.TSD = 1 and CPL != 0.
#UD	If CPUID.07H.00H:ECX.WAITPKG[5] = 0.

UNPCKHPD—Unpack and Interleave High Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 15 /r UNPCKHPD xmm1, xmm2/m128	A	V/V	SSE2	Unpacks and Interleaves double precision floating-point values from high quadwords of xmm1 and xmm2/m128.
VEX.128.66.0F.WIG 15 /r VUNPCKHPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves double precision floating-point values from high quadwords of xmm2 and xmm3/m128.
VEX.256.66.0F.WIG 15 /r VUNPCKHPD ymm1,ymm2, ymm3/m256	B	V/V	AVX	Unpacks and Interleaves double precision floating-point values from high quadwords of ymm2 and ymm3/m256.
EVEX.128.66.0F.W1 15 /r VUNPCKHPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Unpacks and Interleaves double precision floating-point values from high quadwords of xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.256.66.0F.W1 15 /r VUNPCKHPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Unpacks and Interleaves double precision floating-point values from high quadwords of ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.512.66.0F.W1 15 /r VUNPCKHPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Unpacks and Interleaves double precision floating-point values from high quadwords of zmm2 and zmm3/m512/m64bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs an interleaved unpack of the high double precision floating-point values from the first source operand and the second source operand. See Figure 4-15 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is a XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Operation

VUNPCKHPD (EVEX Encoded Versions When SRC2 is a Register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF VL >= 128
    TMP_DEST[63:0] := SRC1[127:64]
    TMP_DEST[127:64] := SRC2[127:64]
FI;
IF VL >= 256
    TMP_DEST[191:128] := SRC1[255:192]
    TMP_DEST[255:192] := SRC2[255:192]
FI;
IF VL >= 512
    TMP_DEST[319:256] := SRC1[383:320]
    TMP_DEST[383:320] := SRC2[383:320]
    TMP_DEST[447:384] := SRC1[511:448]
    TMP_DEST[511:448] := SRC2[511:448]
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VUNPCKHPD (EVEX Encoded Version When SRC2 is Memory)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i]
    FI;
ENDFOR;
IF VL >= 128
    TMP_DEST[63:0] := SRC1[127:64]
    TMP_DEST[127:64] := TMP_SRC2[127:64]
FI;
IF VL >= 256
    TMP_DEST[191:128] := SRC1[255:192]
    TMP_DEST[255:192] := TMP_SRC2[255:192]
FI;
IF VL >= 512
    TMP_DEST[319:256] := SRC1[383:320]
    TMP_DEST[383:320] := TMP_SRC2[383:320]
    TMP_DEST[447:384] := SRC1[511:448]
    TMP_DEST[511:448] := TMP_SRC2[511:448]
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VUNPCKHPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[127:64]
DEST[127:64] := SRC2[127:64]
DEST[191:128] := SRC1[255:192]
DEST[255:192] := SRC2[255:192]
DEST[MAXVL-1:256] := 0
```

VUNPCKHPD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[127:64]
DEST[127:64] := SRC2[127:64]
DEST[MAXVL-1:128] := 0
```

UNPCKHPD (128-bit Legacy SSE Version)

```
DEST[63:0] := SRC1[127:64]
DEST[127:64] := SRC2[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VUNPCKHPD __m512d _mm512_unpackhi_pd(__m512d a, __m512d b);
VUNPCKHPD __m512d _mm512_mask_unpackhi_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VUNPCKHPD __m512d _mm512_maskz_unpackhi_pd(__mmask8 k, __m512d a, __m512d b);
VUNPCKHPD __m256d _mm256_unpackhi_pd(__m256d a, __m256d b)
VUNPCKHPD __m256d _mm256_mask_unpackhi_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VUNPCKHPD __m256d _mm256_maskz_unpackhi_pd(__mmask8 k, __m256d a, __m256d b);
UNPCKHPD __m128d _mm_unpackhi_pd(__m128d a, __m128d b)
VUNPCKHPD __m128d _mm_mask_unpackhi_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VUNPCKHPD __m128d _mm_maskz_unpackhi_pd(__mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instructions, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-52, "Type E4NF Class Exception Conditions."

UNPCKHPS—Unpack and Interleave High Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 15 /r UNPCKHPS xmm1, xmm2/m128	A	V/V	SSE	Unpacks and Interleaves single precision floating-point values from high quadwords of xmm1 and xmm2/m128.
VEX.128.0F.WIG 15 /r VUNPCKHPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves single precision floating-point values from high quadwords of xmm2 and xmm3/m128.
VEX.256.0F.WIG 15 /r VUNPCKHPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Unpacks and Interleaves single precision floating-point values from high quadwords of ymm2 and ymm3/m256.
EVEX.128.0F.W0 15 /r VUNPCKHPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Unpacks and Interleaves single precision floating-point values from high quadwords of xmm2 and xmm3/m128/m32bcst and write result to xmm1 subject to writemask k1.
EVEX.256.0F.W0 15 /r VUNPCKHPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Unpacks and Interleaves single precision floating-point values from high quadwords of ymm2 and ymm3/m256/m32bcst and write result to ymm1 subject to writemask k1.
EVEX.512.0F.W0 15 /r VUNPCKHPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Unpacks and Interleaves single precision floating-point values from high quadwords of zmm2 and zmm3/m512/m32bcst and write result to zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs an interleaved unpack of the high single precision floating-point values from the first source operand and the second source operand.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers.

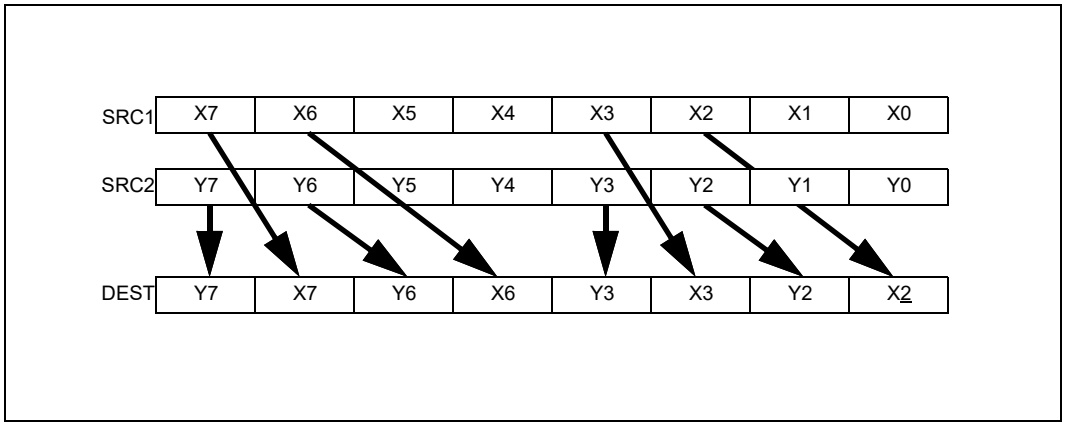


Figure 4-27. VUNPCKHPS Operation

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is a XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Operation

VUNPCKHPS (EVEX Encoded Version When SRC2 is a Register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL >= 128
    TMP_DEST[31:0] := SRC1[95:64]
    TMP_DEST[63:32] := SRC2[95:64]
    TMP_DEST[95:64] := SRC1[127:96]
    TMP_DEST[127:96] := SRC2[127:96]
FI;
IF VL >= 256
    TMP_DEST[159:128] := SRC1[223:192]
    TMP_DEST[191:160] := SRC2[223:192]
    TMP_DEST[223:192] := SRC1[255:224]
    TMP_DEST[255:224] := SRC2[255:224]
FI;
IF VL >= 512
    TMP_DEST[287:256] := SRC1[351:320]
    TMP_DEST[319:288] := SRC2[351:320]
    TMP_DEST[351:320] := SRC1[383:352]
    TMP_DEST[383:352] := SRC2[383:352]
    TMP_DEST[415:384] := SRC1[479:448]
    TMP_DEST[447:416] := SRC2[479:448]
    TMP_DEST[479:448] := SRC1[511:480]
    TMP_DEST[511:480] := SRC2[511:480]
FI;
```

```
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VUNPCKHPS (EVEX Encoded Version When SRC2 is Memory)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+31:i] := SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i]
    FI;
ENDFOR;
IF VL >= 128
    TMP_DEST[31:0] := SRC1[95:64]
    TMP_DEST[63:32] := TMP_SRC2[95:64]
    TMP_DEST[95:64] := SRC1[127:96]
    TMP_DEST[127:96] := TMP_SRC2[127:96]
FI;
IF VL >= 256
    TMP_DEST[159:128] := SRC1[223:192]
    TMP_DEST[191:160] := TMP_SRC2[223:192]
    TMP_DEST[223:192] := SRC1[255:224]
    TMP_DEST[255:224] := TMP_SRC2[255:224]
FI;
IF VL >= 512
    TMP_DEST[287:256] := SRC1[351:320]
    TMP_DEST[319:288] := TMP_SRC2[351:320]
    TMP_DEST[351:320] := SRC1[383:352]
    TMP_DEST[383:352] := TMP_SRC2[383:352]
    TMP_DEST[415:384] := SRC1[479:448]
    TMP_DEST[447:416] := TMP_SRC2[479:448]
    TMP_DEST[479:448] := SRC1[511:480]
    TMP_DEST[511:480] := TMP_SRC2[511:480]
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
```

```
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VUNPCKHPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[95:64]
DEST[63:32] := SRC2[95:64]
DEST[95:64] := SRC1[127:96]
DEST[127:96] := SRC2[127:96]
DEST[159:128] := SRC1[223:192]
DEST[191:160] := SRC2[223:192]
DEST[223:192] := SRC1[255:224]
DEST[255:224] := SRC2[255:224]
DEST[MAXVL-1:256] := 0
```

VUNPCKHPS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[95:64]
DEST[63:32] := SRC2[95:64]
DEST[95:64] := SRC1[127:96]
DEST[127:96] := SRC2[127:96]
DEST[MAXVL-1:128] := 0
```

UNPCKHPS (128-bit Legacy SSE Version)

```
DEST[31:0] := SRC1[95:64]
DEST[63:32] := SRC2[95:64]
DEST[95:64] := SRC1[127:96]
DEST[127:96] := SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VUNPCKHPS __m512 _mm512_unpackhi_ps(__m512 a, __m512 b);
VUNPCKHPS __m512 _mm512_mask_unpackhi_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VUNPCKHPS __m512 _mm512_maskz_unpackhi_ps(__mmask16 k, __m512 a, __m512 b);
VUNPCKHPS __m256 _mm256_unpackhi_ps (__m256 a, __m256 b);
VUNPCKHPS __m256 _mm256_mask_unpackhi_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VUNPCKHPS __m256 _mm256_maskz_unpackhi_ps(__mmask8 k, __m256 a, __m256 b);
UNPCKHPS __m128 _mm_unpackhi_ps (__m128 a, __m128 b);
VUNPCKHPS __m128 _mm_mask_unpackhi_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VUNPCKHPS __m128 _mm_maskz_unpackhi_ps(__mmask8 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-52, “Type E4NF Class Exception Conditions.”

UNPCKLPD—Unpack and Interleave Low Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 14 /r UNPCKLPD xmm1, xmm2/m128	A	V/V	SSE2	Unpacks and Interleaves double precision floating-point values from low quadwords of xmm1 and xmm2/m128.
VEX.128.66.0F.WIG 14 /r VUNPCKLPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves double precision floating-point values from low quadwords of xmm2 and xmm3/m128.
VEX.256.66.0F.WIG 14 /r VUNPCKLPD ymm1,ymm2, ymm3/m256	B	V/V	AVX	Unpacks and Interleaves double precision floating-point values from low quadwords of ymm2 and ymm3/m256.
EVEX.128.66.0F.W1 14 /r VUNPCKLPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Unpacks and Interleaves double precision floating-point values from low quadwords of xmm2 and xmm3/m128/m64bcst subject to write mask k1.
EVEX.256.66.0F.W1 14 /r VUNPCKLPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Unpacks and Interleaves double precision floating-point values from low quadwords of ymm2 and ymm3/m256/m64bcst subject to write mask k1.
EVEX.512.66.0F.W1 14 /r VUNPCKLPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Unpacks and Interleaves double precision floating-point values from low quadwords of zmm2 and zmm3/m512/m64bcst subject to write mask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs an interleaved unpack of the low double precision floating-point values from the first source operand and the second source operand.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Operation

VUNPCKLPD (EVEX Encoded Versions When SRC2 is a Register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF VL >= 128
    TMP_DEST[63:0] := SRC1[63:0]
    TMP_DEST[127:64] := SRC2[63:0]
FI;
IF VL >= 256
    TMP_DEST[191:128] := SRC1[191:128]
    TMP_DEST[255:192] := SRC2[191:128]
FI;
IF VL >= 512
    TMP_DEST[319:256] := SRC1[319:256]
    TMP_DEST[383:320] := SRC2[319:256]
    TMP_DEST[447:384] := SRC1[447:384]
    TMP_DEST[511:448] := SRC2[447:384]
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VUNPCKLPD (EVEX Encoded Version When SRC2 is Memory)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i]
    FI;
ENDFOR;
IF VL >= 128
    TMP_DEST[63:0] := SRC1[63:0]
    TMP_DEST[127:64] := TMP_SRC2[63:0]
FI;
IF VL >= 256
    TMP_DEST[191:128] := SRC1[191:128]
    TMP_DEST[255:192] := TMP_SRC2[191:128]
FI;
IF VL >= 512
    TMP_DEST[319:256] := SRC1[319:256]
    TMP_DEST[383:320] := TMP_SRC2[319:256]
    TMP_DEST[447:384] := SRC1[447:384]
    TMP_DEST[511:448] := TMP_SRC2[447:384]
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VUNPCKLPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[191:128] := SRC1[191:128]
DEST[255:192] := SRC2[191:128]
DEST[MAXVL-1:256] := 0
```

VUNPCKLPD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[MAXVL-1:128] := 0
```

UNPCKLPD (128-bit Legacy SSE Version)

```
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VUNPCKLPD __m512d _mm512_unpacklo_pd(__m512d a, __m512d b);
VUNPCKLPD __m512d _mm512_mask_unpacklo_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VUNPCKLPD __m512d _mm512_maskz_unpacklo_pd(__mmask8 k, __m512d a, __m512d b);
VUNPCKLPD __m256d _mm256_unpacklo_pd(__m256d a, __m256d b)
VUNPCKLPD __m256d _mm256_mask_unpacklo_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VUNPCKLPD __m256d _mm256_maskz_unpacklo_pd(__mmask8 k, __m256d a, __m256d b);
UNPCKLPD __m128d _mm_unpacklo_pd(__m128d a, __m128d b)
VUNPCKLPD __m128d _mm_mask_unpacklo_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VUNPCKLPD __m128d _mm_maskz_unpacklo_pd(__mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-52, “Type E4NF Class Exception Conditions.”

UNPCKLPS—Unpack and Interleave Low Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 14 /r UNPCKLPS xmm1, xmm2/m128	A	V/V	SSE	Unpacks and Interleaves single precision floating-point values from low quadwords of xmm1 and xmm2/m128.
VEX.128.0F.WIG 14 /r VUNPCKLPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves single precision floating-point values from low quadwords of xmm2 and xmm3/m128.
VEX.256.0F.WIG 14 /r VUNPCKLPS ymm1,ymm2,ymm3/m256	B	V/V	AVX	Unpacks and Interleaves single precision floating-point values from low quadwords of ymm2 and ymm3/m256.
EVEX.128.0F.W0 14 /r VUNPCKLPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Unpacks and Interleaves single precision floating-point values from low quadwords of xmm2 and xmm3/mem and write result to xmm1 subject to write mask k1.
EVEX.256.0F.W0 14 /r VUNPCKLPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Unpacks and Interleaves single precision floating-point values from low quadwords of ymm2 and ymm3/mem and write result to ymm1 subject to write mask k1.
EVEX.512.0F.W0 14 /r VUNPCKLPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Unpacks and Interleaves single precision floating-point values from low quadwords of zmm2 and zmm3/m512/m32bcst and write result to zmm1 subject to write mask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs an interleaved unpack of the low single precision floating-point values from the first source operand and the second source operand.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

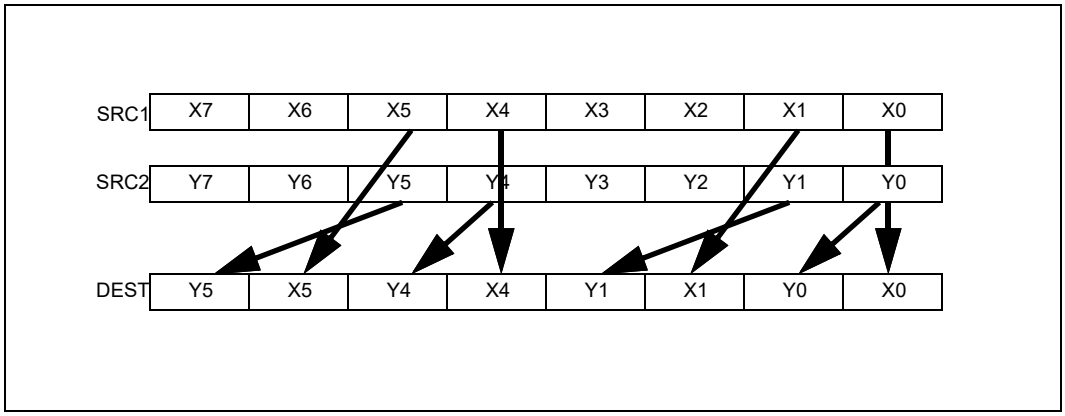


Figure 4-28. VUNPCKLPS Operation

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Operation

VUNPCKLPS (EVEX Encoded Version When SRC2 is a ZMM Register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL >= 128
    TMP_DEST[31:0] := SRC1[31:0]
    TMP_DEST[63:32] := SRC2[31:0]
    TMP_DEST[95:64] := SRC1[63:32]
    TMP_DEST[127:96] := SRC2[63:32]
FI;
IF VL >= 256
    TMP_DEST[159:128] := SRC1[159:128]
    TMP_DEST[191:160] := SRC2[159:128]
    TMP_DEST[223:192] := SRC1[191:160]
    TMP_DEST[255:224] := SRC2[191:160]
FI;
IF VL >= 512
    TMP_DEST[287:256] := SRC1[287:256]
    TMP_DEST[319:288] := SRC2[287:256]
    TMP_DEST[351:320] := SRC1[319:288]
    TMP_DEST[383:352] := SRC2[319:288]
    TMP_DEST[415:384] := SRC1[415:384]
    TMP_DEST[447:416] := SRC2[415:384]
    TMP_DEST[479:448] := SRC1[447:416]
    TMP_DEST[511:480] := SRC2[447:416]
FI;
FOR j := 0 TO KL-1
```

```
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VUNPCKLPS (EVEX Encoded Version When SRC2 is Memory)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 31
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+31:i] := SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i]
    FI;
ENDFOR;
IF VL >= 128
TMP_DEST[31:0] := SRC1[31:0]
TMP_DEST[63:32] := TMP_SRC2[31:0]
TMP_DEST[95:64] := SRC1[63:32]
TMP_DEST[127:96] := TMP_SRC2[63:32]
FI;
IF VL >= 256
    TMP_DEST[159:128] := SRC1[159:128]
    TMP_DEST[191:160] := TMP_SRC2[159:128]
    TMP_DEST[223:192] := SRC1[191:160]
    TMP_DEST[255:224] := TMP_SRC2[191:160]
FI;
IF VL >= 512
    TMP_DEST[287:256] := SRC1[287:256]
    TMP_DEST[319:288] := TMP_SRC2[287:256]
    TMP_DEST[351:320] := SRC1[319:288]
    TMP_DEST[383:352] := TMP_SRC2[319:288]
    TMP_DEST[415:384] := SRC1[415:384]
    TMP_DEST[447:416] := TMP_SRC2[415:384]
    TMP_DEST[479:448] := SRC1[447:416]
    TMP_DEST[511:480] := TMP_SRC2[447:416]
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
```

FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

UNPCKLPS (VEX.256 Encoded Version)
DEST[31:0] := SRC1[31:0]
DEST[63:32] := SRC2[31:0]
DEST[95:64] := SRC1[63:32]
DEST[127:96] := SRC2[63:32]
DEST[159:128] := SRC1[159:128]
DEST[191:160] := SRC2[159:128]
DEST[223:192] := SRC1[191:160]
DEST[255:224] := SRC2[191:160]
DEST[MAXVL-1:256] := 0

VUNPCKLPS (VEX.128 Encoded Version)
DEST[31:0] := SRC1[31:0]
DEST[63:32] := SRC2[31:0]
DEST[95:64] := SRC1[63:32]
DEST[127:96] := SRC2[63:32]
DEST[MAXVL-1:128] := 0

UNPCKLPS (128-bit Legacy SSE Version)
DEST[31:0] := SRC1[31:0]
DEST[63:32] := SRC2[31:0]
DEST[95:64] := SRC1[63:32]
DEST[127:96] := SRC2[63:32]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VUNPCKLPS __m512 _mm512_unpacklo_ps(__m512 a, __m512 b);
VUNPCKLPS __m512 _mm512_mask_unpacklo_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VUNPCKLPS __m512 _mm512_maskz_unpacklo_ps(__mmask16 k, __m512 a, __m512 b);
VUNPCKLPS __m256 _mm256_unpacklo_ps (__m256 a, __m256 b);
VUNPCKLPS __m256 _mm256_mask_unpacklo_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VUNPCKLPS __m256 _mm256_maskz_unpacklo_ps(__mmask8 k, __m256 a, __m256 b);
UNPCKLPS __m128 _mm_unpacklo_ps (__m128 a, __m128 b);
VUNPCKLPS __m128 _mm_mask_unpacklo_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VUNPCKLPS __m128 _mm_maskz_unpacklo_ps(__mmask8 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-52, “Type E4NF Class Exception Conditions.”

URDMSR—User Read from Model-Specific Register

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 38 F8 11:rrr:bbb URDMSR r64, r64	MR	V/N.E.	USER_MSR	Load into register bbb the value of the MSR with address in rrr.
VEX.128.F2.MAP7:W0.F8 11:000:bbb URDMSR r64, imm32	MI	V/N.E.	USER_MSR	Load into register bbb the value of the MSR with address in the 32-bit immediate.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r)	ModRM:reg (r)	N/A	N/A
MI	ModRM:r/m (r)	imm32	N/A	N/A

Description

URDMSR reads the contents of a 64-bit MSR specified in operand 2 into operand 1. Operand 1 is a general-purpose register, while operand 2 may be either a general-purpose register or an immediate. URDMSR reads the indicated MSR in the same manner as RDMSR.

MSRs readable by RDMSR with CPL = 0 can be read by URDMSR at any privilege level but under OS control. The OS controls what MSRs can be read by the URDMSR and UWRMSR instructions with a 4-KByte **user-MSR bitmap** located at an aligned linear address in the IA32_USER_MSR_CTL MSR (address 1CH). The URDMSR instruction is enabled only if bit 0 of this MSR is 1.

The low 2 KBytes of the bitmap control URDMSR (they compose the URDMSR bitmap); these 2 KBytes include one bit for each MSR address in the range 0H–3FFFH. URDMSR may read an MSR only if the bit corresponding to the MSR has value 1; otherwise (or if the MSR address is outside that range) URDMSR causes a general-protection exception (#GP).

The URDMSR accesses to these bitmaps are implicit supervisor-mode accesses, which means they use supervisor privilege regardless of CPL.

Operation

```
DEST := MSR[SRC]
```

Flags Affected

None.

Protected Mode Exceptions

#UD	The URDMSR instruction is not recognized outside 64-bit mode.

Real-Address Mode Exceptions

#UD	The URDMSR instruction is not recognized outside 64-bit mode.

Virtual-8086 Mode Exceptions

#UD	The URDMSR instruction is not recognized outside 64-bit mode.

Compatibility Mode Exceptions

#UD	The URDMSR instruction is not recognized outside 64-bit mode.

64-Bit Mode Exceptions

#GP(0)	If MSR_address[63:14] is not all zero.

	If the MSR address is in the range 0—3FFH and bit n in the URDMSR bitmap is 0, where n is the MSR address.
	If RDMSR from the specified MSR would result in a #GP(0) exception due to the MSR not being accessible.
#UD	If the LOCK prefix is used.
	If CPUID.07H.01H:EDX.USER_MSR[15] = 0.
	If IA32_USER_MSR_CTL.ENABLE = 0.

UWRMSR—User Write to Model-Specific Register

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 38 F8 11:rrr:bbb UWRMSR r64, r64	RM	V/N.E.	USER_MSR	Write the value of register bbb to the MSR with address in rrr.
VEX.128.F3.MAP7:W0.F8 11:000:bbb UWRMSR imm32, r64	IM	V/N.E.	USER_MSR	Write the value of register bbb to the MSR specified in the 32-bit immediate.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
IM	imm32	ModRM:r/m (r)	N/A	N/A

Description

UWRMSR writes the contents of operand 2 into the 64-bit MSR specified in operand 1. Operand 2 is a general-purpose register, while operand 1 may be either a general-purpose register or an immediate. UWRMSR writes the indicated MSR in the same manner as WRMSR, but it is limited to a specific set of MSRs. Currently, the only such MSR is the IA32_UARCH_MISC_CTL MSR (address 1B01H).

The indicated MSRs can be written by UWRMSR at any privilege level but under OS control. The OS controls what MSRs can be read by the URDMSR and UWRMSR instructions with a 4-KByte **user-MSR bitmap** located at an aligned linear address in the IA32_USER_MSR_CTL MSR (address 1CH). The UWRMSR instruction is enabled only if bit 0 of this MSR is 1.

The high 2 KBytes of the bitmap control UWRMSR (they compose the UWRMSR bitmap); these 2 KBytes include one bit for each MSR address in the range 0H–3FFFH. UWRMSR may write to an MSR only if the bit corresponding to the MSR has value 1; otherwise (or if the MSR address is outside that range) UWRMSR causes a general-protection exception (#GP).

UWRMSR accesses to these bitmaps are implicit supervisor-mode accesses, which means they use supervisor privilege regardless of CPL. UWRMSR behaves like WRMSRNS and is not defined as a serializing instruction (see "Serializing Instructions" in Chapter 11 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A). Refer to the WRMSRNS instruction for a thorough explanation of what this implies.

Operation

```
MSR[DEST] := SRC
```

Flags Affected

None.

Protected Mode Exceptions

#UD	The UWRMSR instruction is not recognized outside 64-bit mode.

Real-Address Mode Exceptions

#UD	The UWRMSR instruction is not recognized outside 64-bit mode.

Virtual-8086 Mode Exceptions

#UD	The UWRMSR instruction is not recognized outside 64-bit mode.

Compatibility Mode Exceptions

#UD	The UWRMSR instruction is not recognized outside 64-bit mode.

64-Bit Mode Exceptions

#GP(0)	If MSR_address[63:14] is not all zero.
	If the MSR address is in the range 0—3FFH and bit n in UWRMSR bitmap is 0, where n is the MSR address.
	If an MSR other than IA32_UARCH_MISC_CTL is specified.
	If WRMSR to the specified MSR would result in a #GP(0) exception due to the MSR not being accessible or attempting to set bits that are reserved.
#UD	If the LOCK prefix is used.
	If CPUID.07H.01H:EDX.USER_MSR[15] = 0.
	If IA32_USER_MSR_CTL.ENABLE = 0.

CHAPTER 5
INSTRUCTION SET REFERENCE, V

5.1 TERNARY BIT VECTOR LOGIC TABLE

VPTERNLOGD/VPTERNLOGQ instructions operate on dword/qword elements and take three bit vectors of the respective input data elements to form a set of 32/64 indices, where each 3-bit value provides an index into an 8-bit lookup table represented by the imm8 byte of the instruction. The 256 possible values of the imm8 byte is constructed as a 16x16 boolean logic table. The 16 rows of the table uses the lower 4 bits of imm8 as row index. The 16 columns are referenced by imm8[7:4]. The 16 columns of the table are present in two halves, with 8 columns shown in Table 5-1 for the column index value between 0:7, followed by Table 5-2 showing the 8 columns corresponding to column index 8:15. This section presents the two-halves of the 256-entry table using a short-hand notation representing simple or compound boolean logic expressions with three input bit source data.

The three input bit source data will be denoted with the capital letters: A, B, C; where A represents a bit from the first source operand (also the destination operand), B and C represent a bit from the 2nd and 3rd source operands.

Each map entry takes the form of a logic expression consisting of one of more component expressions. Each component expression consists of either a unary or binary boolean operator and associated operands. Each binary boolean operator is expressed in lowercase letters, and operands concatenated after the logic operator. The unary operator 'not' is expressed using '!'. Additionally, the conditional expression "A?B:C" expresses a result returning B if A is set, returning C otherwise.

A binary boolean operator is followed by two operands, e.g., andAB. For a compound binary expression that contain commutative components and comprising the same logic operator, the 2nd logic operator is omitted and three operands can be concatenated in sequence, e.g., andABC. When the 2nd operand of the first binary boolean expression comes from the result of another boolean expression, the 2nd boolean expression is concatenated after the uppercase operand of the first logic expression, e.g., norBnandAC. When the result is independent of an operand, that operand is omitted in the logic expression, e.g., zeros or norCB.

The 3-input expression "majorABC" returns 0 if two or more input bits are 0, returns 1 if two or more input bits are 1. The 3-input expression "minorABC" returns 1 if two or more input bits are 0, returns 0 if two or more input bits are 1.

The building-block bit logic functions used in Table 5-1 and Table 5-2 include:

- Constants: TRUE (1), FALSE (0);
- Unary function: Not (!);
- Binary functions: and, nand, or, nor, xor, xnor;
- Conditional function: Select (?:);
- Tertiary functions: major, minor.

Table 5-1. Lower 8 columns of the 16x16 Map of VPTERNLOG Boolean Logic Operations

Imm	[7:4]							
[3:0]	0H	1H	2H	3H	4H	5H	6H	7H
00H	FALSE	andAnorBC	norBnandAC	andA!B	norCnandBA	andA!C	andAxorBC	andAnandBC
01H	norABC	norCB	norBxorAC	A?!B:norBC	norCxorBA	A?!C:norBC	A?xorBC:norBC	A?nandBC:norBC
02H	andCnorBA	norBxnorAC	andC!B	norBnorAC	C?norBA:andBA	C?norBA:A	C?!B:andBA	C?!B:A
03H	norBA	norBandAC	C?!B:norBA	!B	C?norBA:xnorBA	A?!C:!B	A?xorBC:!B	A?nandBC:!B
04H	andBnorAC	norCxnorBA	B?norAC:andAC	B?norAC:A	andB!C	norCnorBA	B?!C:andAC	B?!C:A
05H	norCA	norCandBA	B?norAC:xnorAC	A?!B:!C	B?!C:norAC	!C	A?xorBC:!C	A?nandBC:!C
06H	norAxnorBC	A?norBC:xorBC	B?norAC:C	xorBorAC	C?norBA:B	xorCorBA	xorCB	B?!C:orAC
07H	norAandBC	minorABC	C?!B:!A	nandBorAC	B?!C:!A	nandCorBA	A?xorBC:nandBC	nandCB
08H	norAnandBC	A?norBC:andBC	andCxorBA	A?!B:andBC	andBxorAC	A?!C:andBC	A?xorBC:andBC	xorAandBC
09H	norAxorBC	A?norBC:xnorBC	C?xorBA:norBA	A?!B:xnorBC	B?xorAC:norAC	A?!C:xnorBC	xnorABC	A?nandBC:xnorBC
0AH	andC!A	A?norBC:C	andCnandBA	A?!B:C	C?!A:andBA	xorCA	xorCandBA	A?nandBC:C
0BH	C?!A:norBA	C?!A:!B	C?nandBA:norBA	C?nandBA:!B	B?xorAC:!A	B?xorAC:nandAC	C?nandBA:xnorBA	nandBxnorAC
0CH	andB!A	A?norBC:B	B?!A:andAC	xorBA	andBnandAC	A?!C:B	xorBandAC	A?nandBC:B
0DH	B?!A:norAC	B?!A:!C	B?!A:xnorAC	C?xorBA:nandBA	B?nandAC:norAC	B?nandAC:!C	B?nandAC:xnorAC	nandCxnorBA
0EH	norAnorBC	xorAorBC	B?!A:C	A?!B:orBC	C?!A:B	A?!C:orBC	B?nandAC:C	A?nandBC:orBC
0FH	!A	nandAorBC	C?nandBA:!A	nandBA	B?nandAC:!A	nandCA	nandAxnorBC	nandABC

Table 5-2 shows the half of 256-entry map corresponding to column index values 8:15.

Table 5-2. Upper 8 columns of the 16x16 Map of VPTERNLOG Boolean Logic Operations

Imm	[7:4]							
[3:0]	08H	09H	0AH	0BH	0CH	0DH	0EH	0FH
00H	andABC	andAxnorBC	andCA	B?andAC:A	andBA	C?andBA:A	andAorBC	A
01H	A?andBC:norBC	B?andAC:!C	A?C:norBC	C?A:!B	A?B:norBC	B?A:!C	xnorAorBC	orAnorBC
02H	andCxnorBA	B?andAC:xorAC	B?andAC:C	B?andAC:orAC	C?xnorBA:andBA	B?A:xorAC	B?A:C	B?A:orAC
03H	A?andBC:!B	xnorBandAC	A?C:!B	nandBnandAC	xnorBA	B?A:nandAC	A?orBC:!B	orA!B
04H	andBxnorAC	C?andBA:xorBA	B?xnorAC:andAC	B?xnorAC:A	C?andBA:B	C?andBA:orBA	C?A:B	C?A:orBA
05H	A?andBC:!C	xnorCandBA	xnorCA	C?A:nandBA	A?B:!C	nandCnandBA	A?orBC:!C	orA!C
06H	A?andBC:xorBC	xorABC	A?C:xorBC	B?xnorAC:orAC	A?B:xorBC	C?xnorBA:orBA	A?orBC:xorBC	orAxorBC
07H	xnorAandBC	A?xnorBC:nandBC	A?C:nandBC	nandBxorAC	A?B:nandBC	nandCxorBA	A?orBCnandBC	orAnandBC
08H	andCB	A?xnorBC:andBC	andCorAB	B?C:A	andBorAC	C?B:A	majorABC	orAandBC
09H	B?C:norAC	xnorCB	xnorCorBA	C?orBA:!B	xnorBorAC	B?orAC:!C	A?orBC:xnorBC	orAxnorBC
0AH	A?andBC:C	A?xnorBC:C	C	B?C:orAC	A?B:C	B?orAC:xorAC	orCandBA	orCA
0BH	B?C:!A	B?C:nandAC	orCnorBA	orC!B	B?orAC:!A	B?orAC:nandAC	orCxnorBA	nandBnorAC
0CH	A?andBC:B	A?xnorBC:B	A?C:B	C?orBA:xorBA	B	C?B:orBA	orBandAC	orBA
0DH	C?B!A	C?B:nandBA	C?orBA:!A	C?orBA:nandBA	orBnorAC	orB!C	orBxnorAC	nandCnorBA
0EH	A?andBC:orBC	A?xnorBC:orBC	A?C:orBC	orCxorBA	A?B:orBC	orBxorAC	orCB	orABC
0FH	nandAnandBC	nandAxorBC	orC!A	orCnandBA	orB!A	orBnandAC	nandAnorBC	TRUE

Table 5-1 and Table 5-2 translate each of the possible value of the imm8 byte to a Boolean expression. These tables can also be used by software to translate Boolean expressions to numerical constants to form the imm8 value needed to construct the VPTERNLOG syntax. There is a unique set of three byte constants (F0H, CCH, AAH) that can be used for this purpose as input operands in conjunction with the Boolean expressions defined in those tables. The reverse mapping can be expressed as:

Result_imm8 = Table_Lookup_Entry(0F0H, 0CCH, 0AAH)

Table_Lookup_Entry is the Boolean expression defined in Table 5-1 and Table 5-2.

5.2 INSTRUCTIONS (V)

Chapter 5 continues an alphabetical discussion of Intel® 64 and IA-32 instructions (V). See also: Chapter 3, "Instruction Set Reference, A-L," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume

2A; Chapter 5, “Instruction Set Reference, V,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B; and Chapter 5, “Instruction Set Reference, V,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D.

VADDPH—Add Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 58 /r VADDPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Add packed FP16 value from xmm3/m128/m16bcst to xmm2, and store result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 58 /r VADDPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Add packed FP16 value from ymm3/m256/m16bcst to ymm2, and store result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 58 /r VADDPH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Add packed FP16 value from zmm3/m512/m16bcst to zmm2, and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction adds packed FP16 values from source operands and stores the packed FP16 result in the destination operand. The destination elements are updated according to the writemask.

Operation

VADDPH (EVEX Encoded Versions) When SRC2 Operand is a Register

```
VL = 128, 256 or 512
KL := VL/16
IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.fp16[j] := SRC1.fp16[j] + SRC2.fp16[j]
    ELSEIF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged
DEST[MAXVL-1:VL] := 0
```

VADDPH (EVEX Encoded Versions) When SRC2 Operand is a Memory Source

```
VL = 128, 256 or 512
KL := VL/16
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            DEST.fp16[j] := SRC1.fp16[j] + SRC2.fp16[0]
        ELSE:
            DEST.fp16[j] := SRC1.fp16[j] + SRC2.fp16[j]
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VADDPH __m128h _mm_add_ph (__m128h a, __m128h b);
VADDPH __m128h _mm_mask_add_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VADDPH __m128h _mm_maskz_add_ph (__mmask8 k, __m128h a, __m128h b);
VADDPH __m256h _mm256_add_ph (__m256h a, __m256h b);
VADDPH __m256h _mm256_mask_add_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VADDPH __m256h _mm256_maskz_add_ph (__mmask16 k, __m256h a, __m256h b);
VADDPH __m512h _mm512_add_ph (__m512h a, __m512h b);
VADDPH __m512h _mm512_add_ph (__m512h a, __m512h b);
VADDPH __m512h _mm512_mask_add_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VADDPH __m512h _mm512_maskz_add_ph (__mmask32 k, __m512h a, __m512h b);
VADDPH __m512h _mm512_add_round_ph (__m512h a, __m512h b, int rounding);
VADDPH __m512h _mm512_mask_add_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int rounding);
VADDPH __m512h _mm512_maskz_add_round_ph (__mmask32 k, __m512h a, __m512h b, int rounding);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VADDSH—Add Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 58 /r VADDSH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512_FP16 OR AVX10.1	Add the low FP16 value from xmm3/m16 to xmm2, and store the result in xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction adds the low FP16 value from the source operands and stores the FP16 result in the destination operand.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

VADDSH (EVEX Encoded Versions)

```
IF EVEX.b = 1 and SRC2 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)
IF k1[0] OR *no writemask*:
    DEST.fp16[0] := SRC1.fp16[0] + SRC2.fp16[0]
ELSEIF *zeroing*:
    DEST.fp16[0] := 0
// else dest.fp16[0] remains unchanged
DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VADDSH __m128h _mm_add_round_sh (__m128h a, __m128h b, int rounding);
VADDSH __m128h _mm_mask_add_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int rounding);
VADDSH __m128h _mm_maskz_add_round_sh (__mmask8 k, __m128h a, __m128h b, int rounding);
VADDSH __m128h _mm_add_sh (__m128h a, __m128h b);
VADDSH __m128h _mm_mask_add_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VADDSH __m128h _mm_maskz_add_sh (__mmask8 k, __m128h a, __m128h b);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VALIGND/VALIGNQ—Align Doubleword/Quadword Vectors

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 03 /r ib VALIGND xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift right and merge vectors xmm2 and xmm3/m128/m32bcst with double-word granularity using imm8 as number of elements to shift, and store the final result in xmm1, under writemask.
EVEX.128.66.0F3A.W1 03 /r ib VALIGNQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift right and merge vectors xmm2 and xmm3/m128/m64bcst with quad-word granularity using imm8 as number of elements to shift, and store the final result in xmm1, under writemask.
EVEX.256.66.0F3A.W0 03 /r ib VALIGND ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift right and merge vectors ymm2 and ymm3/m256/m32bcst with double-word granularity using imm8 as number of elements to shift, and store the final result in ymm1, under writemask.
EVEX.256.66.0F3A.W1 03 /r ib VALIGNQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift right and merge vectors ymm2 and ymm3/m256/m64bcst with quad-word granularity using imm8 as number of elements to shift, and store the final result in ymm1, under writemask.
EVEX.512.66.0F3A.W0 03 /r ib VALIGND zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	A	V/V	AVX512F OR AVX10.1	Shift right and merge vectors zmm2 and zmm3/m512/m32bcst with double-word granularity using imm8 as number of elements to shift, and store the final result in zmm1, under writemask.
EVEX.512.66.0F3A.W1 03 /r ib VALIGNQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	A	V/V	AVX512F OR AVX10.1	Shift right and merge vectors zmm2 and zmm3/m512/m64bcst with quad-word granularity using imm8 as number of elements to shift, and store the final result in zmm1, under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Concatenates and shifts right doubleword/quadword elements of the first source operand (the second operand) and the second source operand (the third operand) into a 1024/512/256-bit intermediate vector. The low 512/256/128-bit of the intermediate vector is written to the destination operand (the first operand) using the writemask k1. The destination and first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values (merging-masking) or are set to 0 (zeroing-masking).

Operation

VALIGND (EVEX Encoded Versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
IF (SRC2 *is memory*) (AND EVEX.b = 1)
    THEN
        FOR j := 0 TO KL-1
            i := j * 32
            src[i+31:i] := SRC2[31:0]
        ENDFOR;
    ELSE src := SRC2
FI
; Concatenate sources
tmp[VL-1:0] := src[VL-1:0]
tmp[2VL-1:VL] := SRC1[VL-1:0]
; Shift right doubleword elements
IF VL = 128
    THEN SHIFT = imm8[1:0]
    ELSE
        IF VL = 256
            THEN SHIFT = imm8[2:0]
            ELSE SHIFT = imm8[3:0]
        FI
FI;
tmp[2VL-1:0] := tmp[2VL-1:0] >> (32*SHIFT)
; Apply writemask
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := tmp[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VALIGNQ (EVEX Encoded Versions)

(KL, VL) = (2, 128), (4, 256),(8, 512)

```
IF (SRC2 *is memory*) (AND EVEX.b = 1)
    THEN
        FOR j := 0 TO KL-1
            i := j * 64
            src[i+63:i] := SRC2[63:0]
        ENDFOR;
    ELSE src := SRC2
FI
; Concatenate sources
tmp[VL-1:0] := src[VL-1:0]
tmp[2VL-1:VL] := SRC1[VL-1:0]
; Shift right quadword elements
```

```
IF VL = 128
    THEN SHIFT = imm8[0]
    ELSE
        IF VL = 256
            THEN SHIFT = imm8[1:0]
            ELSE SHIFT = imm8[2:0]
        FI
FI;
tmp[2VL-1:0] := tmp[2VL-1:0] >> (64*SHIFT)
; Apply writemask
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := tmp[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VALIGND __m512i _mm512_alignr_epi32( __m512i a, __m512i b, int cnt);
VALIGND __m512i _mm512_mask_alignr_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b, int cnt);
VALIGND __m512i _mm512_maskz_alignr_epi32( __mmask16 k, __m512i a, __m512i b, int cnt);
VALIGND __m256i _mm256_mask_alignr_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGND __m256i _mm256_maskz_alignr_epi32( __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGND __m128i _mm_mask_alignr_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGND __m128i _mm_maskz_alignr_epi32( __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGNQ __m512i _mm512_alignr_epi64( __m512i a, __m512i b, int cnt);
VALIGNQ __m512i _mm512_mask_alignr_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b, int cnt);
VALIGNQ __m512i _mm512_maskz_alignr_epi64( __mmask8 k, __m512i a, __m512i b, int cnt);
VALIGNQ __m256i _mm256_mask_alignr_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGNQ __m256i _mm256_maskz_alignr_epi64( __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGNQ __m128i _mm_mask_alignr_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGNQ __m128i _mm_maskz_alignr_epi64( __mmask8 k, __m128i a, __m128i b, int cnt);
```

Exceptions

See Table 2-52, "Type E4NF Class Exception Conditions."

VBCSTNEBF162PS—Load BF16 Element and Convert to FP32 Element With Broadcast

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.F3.0F38.W0 B1 !(11):rrr:bbb VBCSTNEBF162PS xmm1, m16	A	V/V	AVX_NE_ CONVERT	Load one BF16 floating-point element from m16, convert to FP32 and store result in xmm1.
VEX.256.F3.0F38.W0 B1 !(11):rrr:bbb VBCSTNEBF162PS ymm1, m16	A	V/V	AVX_NE_ CONVERT	Load one BF16 floating-point element from m16, convert to FP32 and store result in ymm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction loads one BF16 element from memory, converts it to FP32, and broadcasts it to a SIMD register.

This instruction does not generate floating-point exceptions and does not consult or update MXCSR.

Since any BF16 number can be represented in FP32, the conversion result is exact and no rounding is needed.

Operation

VBCSTNEBF162PS dest, src (VEX encoded version)

```
VL = (128, 256)
KL = VL/32

FOR i in range(0, KL):
    tmp.dword[i].word[0] = src.word[0] // reads 16b from memory

FOR i in range(0, KL):
    dest.dword[i] = make_fp32(TMP.dword[i].word[0])
DEST[MAXVL-1:VL] := 0
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VBCSTNEBF162PS __m128 _mm_bcstnebf16_ps (const __bf16* __A);
VBCSTNEBF162PS __m256 _mm256_bcstnebf16_ps (const __bf16* __A);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-22, “Type 5 Class Exception Conditions.”

VBCSTNESH2PS—Load FP16 Element and Convert to FP32 Element with Broadcast

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 B1 !(11):rrr:bbb VBCSTNESH2PS xmm1, m16	A	V/V	AVX_NE_ CONVERT	Load one FP16 element from m16, convert to FP32, and store result in xmm1.
VEX.256.66.0F38.W0 B1 !(11):rrr:bbb VBCSTNESH2PS ymm1, m16	A	V/V	AVX_NE_ CONVERT	Load one FP16 element from m16, convert to FP32, and store result in ymm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction loads one FP16 element from memory, converts it to FP32, and broadcasts it to a SIMD register.

This instruction does not generate floating-point exceptions and does not consult or update MXCSR.

Input FP16 denormals are converted to normal FP32 numbers and not treated as zero. Since any FP16 number can be represented in FP32, the conversion result is exact and no rounding is needed.

Operation

VBCSTNESH2PS dest, src (VEX encoded version)

```
VL = (128, 256)
KL = VL/32

FOR i in range(0, KL):
    tmp.dword[i].word[0] = src.word[0] // read 16b from memory

FOR i in range(0, KL):
    dest.dword[i] = convert_fp16_to_fp32(tmp.dword[i].word[0])  //SAE
DEST[MAXVL-1:VL] := 0
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VBCSTNESH2PS __m128 _mm_bcstnesh_ps (const _Float16* __A);
VBCSTNESH2PS __m256 _mm256_bcstnesh_ps (const _Float16* __A);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-22, “Type 5 Class Exception Conditions.”

VBLENDMPD/VBLENDMPS—Blend Float64/Float32 Vectors Using an OpMask Control

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 65 /r VBLENDMPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Blend double precision vector xmm2 and double precision vector xmm3/m128/m64bcst and store the result in xmm1, under control mask.
EVEX.256.66.0F38.W1 65 /r VBLENDMPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Blend double precision vector ymm2 and double precision vector ymm3/m256/m64bcst and store the result in ymm1, under control mask.
EVEX.512.66.0F38.W1 65 /r VBLENDMPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	A	V/V	AVX512F OR AVX10.1	Blend double precision vector zmm2 and double precision vector zmm3/m512/m64bcst and store the result in zmm1, under control mask.
EVEX.128.66.0F38.W0 65 /r VBLENDMPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Blend single precision vector xmm2 and single precision vector xmm3/m128/m32bcst and store the result in xmm1, under control mask.
EVEX.256.66.0F38.W0 65 /r VBLENDMPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Blend single precision vector ymm2 and single precision vector ymm3/m256/m32bcst and store the result in ymm1, under control mask.
EVEX.512.66.0F38.W0 65 /r VBLENDMPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX512F OR AVX10.1	Blend single precision vector zmm2 and single precision vector zmm3/m512/m32bcst using k1 as select control and store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs an element-by-element blending between float64/float32 elements in the first source operand (the second operand) with the elements in the second source operand (the third operand) using an opmask register as select control. The blended result is written to the destination register.

The destination and first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The opmask register is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for first source operand, 1 for second source operand).

If EVEX.z is set, the elements with corresponding mask bit value of 0 in the destination operand are zeroed.

Operation

VBLENDMPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no controlmask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := SRC2[63:0]
                ELSE
                    DEST[i+63:i] := SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN DEST[i+63:i] := SRC1[i+63:i]
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VBLENDMPS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no controlmask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] := SRC2[31:0]
                ELSE
                    DEST[i+31:i] := SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN DEST[i+31:i] := SRC1[i+31:i]
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VBLENDMPD __m512d _mm512_mask_blend_pd(__mmask8 k, __m512d a, __m512d b);
VBLENDMPD __m256d _mm256_mask_blend_pd(__mmask8 k, __m256d a, __m256d b);
VBLENDMPD __m128d _mm_mask_blend_pd(__mmask8 k, __m128d a, __m128d b);
VBLENDMPS __m512 _mm512_mask_blend_ps(__mmask16 k, __m512 a, __m512 b);
VBLENDMPS __m256 _mm256_mask_blend_ps(__mmask8 k, __m256 a, __m256 b);
VBLENDMPS __m128 _mm_mask_blend_ps(__mmask8 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, "Type E4 Class Exception Conditions."

VBROADCAST—Load with Broadcast Floating-Point Data

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	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.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.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.
EVEX.256.66.0F38.W1 19 /r VBROADCASTSD ymm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Broadcast low double precision floating-point element in xmm2/m64 to four locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 19 /r VBROADCASTSD zmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512F OR AVX10.1	Broadcast low double precision floating-point element in xmm2/m64 to eight locations in zmm1 using writemask k1.
EVEX.256.66.0F38.W0 19 /r VBROADCASTF32X2 ymm1 {k1}{z}, xmm2/m64	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Broadcast two single precision floating-point elements in xmm2/m64 to locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 19 /r VBROADCASTF32X2 zmm1 {k1}{z}, xmm2/m64	C	V/V	AVX512DQ OR AVX10.1	Broadcast two single precision floating-point elements in xmm2/m64 to locations in zmm1 using writemask k1.
EVEX.128.66.0F38.W0 18 /r VBROADCASTSS xmm1 {k1}{z}, xmm2/m32	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Broadcast low single precision floating-point element in xmm2/m32 to all locations in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 18 /r VBROADCASTSS ymm1 {k1}{z}, xmm2/m32	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Broadcast low single precision floating-point element in xmm2/m32 to all locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 18 /r VBROADCASTSS zmm1 {k1}{z}, xmm2/m32	B	V/V	AVX512F OR AVX10.1	Broadcast low single precision floating-point element in xmm2/m32 to all locations in zmm1 using writemask k1.
EVEX.256.66.0F38.W0 1A /r VBROADCASTF32X4 ymm1 {k1}{z}, m128	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Broadcast 128 bits of 4 single precision floating-point data in mem to locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 1A /r VBROADCASTF32X4 zmm1 {k1}{z}, m128	D	V/V	AVX512F OR AVX10.1	Broadcast 128 bits of 4 single precision floating-point data in mem to locations in zmm1 using writemask k1.
EVEX.256.66.0F38.W1 1A /r VBROADCASTF64X2 ymm1 {k1}{z}, m128	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Broadcast 128 bits of 2 double precision floating-point data in mem to locations in ymm1 using writemask k1.

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 1A /r VBROADCASTF64X2 zmm1 {k1}{z}, m128	C	V/V	AVX512DQ OR AVX10.1	Broadcast 128 bits of 2 double precision floating-point data in mem to locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W0 1B /r VBROADCASTF32X8 zmm1 {k1}{z}, m256	E	V/V	AVX512DQ OR AVX10.1	Broadcast 256 bits of 8 single precision floating-point data in mem to locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W1 1B /r VBROADCASTF64X4 zmm1 {k1}{z}, m256	D	V/V	AVX512F OR AVX10.1	Broadcast 256 bits of 4 double precision floating-point data in mem to locations in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
C	Tuple2	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
D	Tuple4	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
E	Tuple8	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

VBROADCASTSD/VBROADCASTSS/VBROADCASTF128 load floating-point values as one tuple from the source operand (second operand) in memory and broadcast to all elements of the destination operand (first operand).

VEX256-encoded versions: The destination operand is a YMM register. The source operand is either a 32-bit, 64-bit, or 128-bit memory location. Register source encodings are reserved and will #UD. Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX-encoded versions: The destination operand is a ZMM/YMM/XMM register and updated according to the write-mask k1. The source operand is either a 32-bit, 64-bit memory location or the low doubleword/quadword element of an XMM register.

VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF64X2/VBROADCASTF32X8/VBROADCASTF64X4 load floating-point values as tuples from the source operand (the second operand) in memory or register and broadcast to all elements of the destination operand (the first operand). The destination operand is a YMM/ZMM register updated according to the writemask k1. The source operand is either a register or 64-bit/128-bit/256-bit memory location.

VBROADCASTSD and VBROADCASTF128,F32x4 and F64x2 are only supported as 256-bit and 512-bit wide versions and up. VBROADCASTSS is supported in 128-bit, 256-bit and 512-bit wide versions. F32x8 and F64x4 are only supported as 512-bit wide versions.

VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF32X8 have 32-bit granularity. VBROADCASTF64X2 and VBROADCASTF64X4 have 64-bit granularity.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

If VBROADCASTSD or VBROADCASTF128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

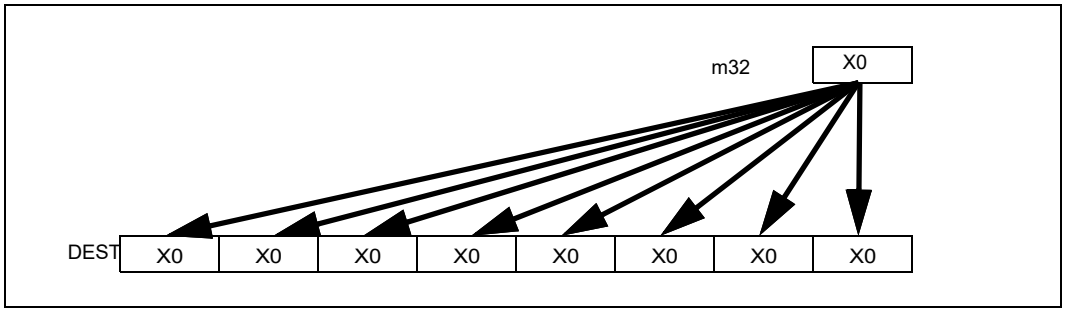


Figure 5-1. VBROADCASTSS Operation (VEX.256 encoded version)

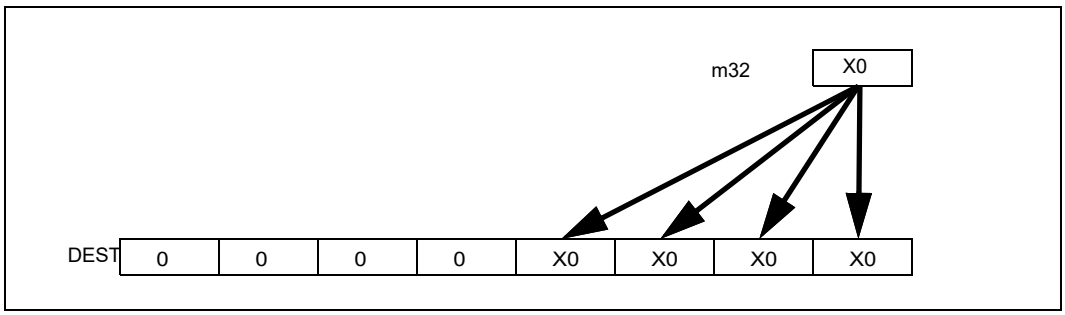


Figure 5-2. VBROADCASTSS Operation (VEX.128-bit version)

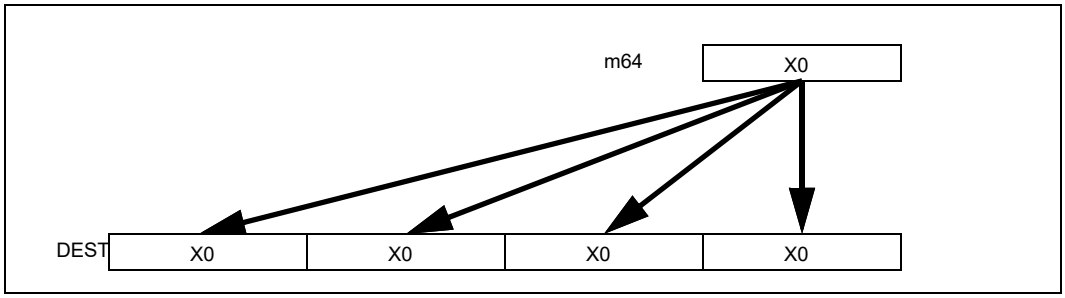


Figure 5-3. VBROADCASTSD Operation (VEX.256-bit version)

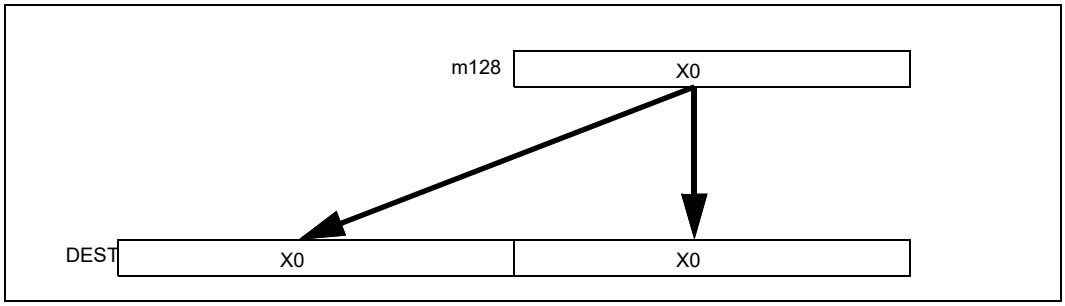


Figure 5-4. VBROADCASTF128 Operation (VEX.256-bit version)

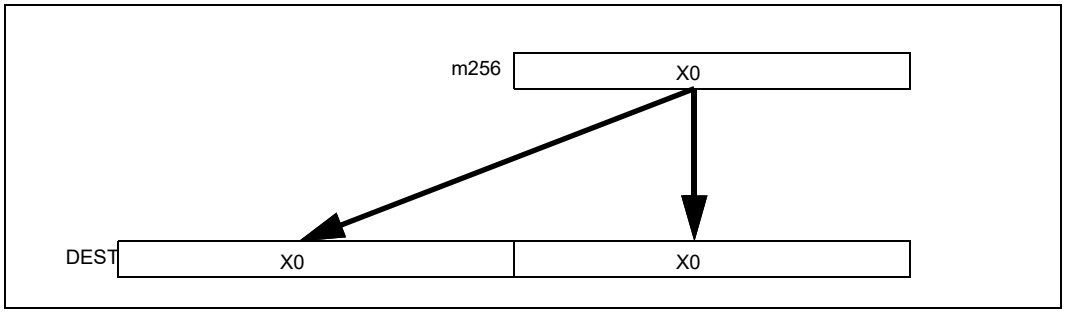


Figure 5-5. VBROADCASTF64X4 Operation (512-bit version with writemask all 1s)

Operation

VBROADCASTSS (128-bit Version VEX and Legacy)

```
temp := SRC[31:0]
DEST[31:0] := temp
DEST[63:32] := temp
DEST[95:64] := temp
DEST[127:96] := temp
DEST[MAXVL-1:128] := 0
```

VBROADCASTSS (VEX.256 Encoded Version)

```
temp := SRC[31:0]
DEST[31:0] := temp
DEST[63:32] := temp
DEST[95:64] := temp
DEST[127:96] := temp
DEST[159:128] := temp
DEST[191:160] := temp
DEST[223:192] := temp
DEST[255:224] := temp
DEST[MAXVL-1:256] := 0
```

VBROADCASTSS (EVEX Encoded Versions)

```
(KL, VL) (4, 128), (8, 256),= (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[31:0]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VBROADCASTSD (VEX.256 Encoded Version)

```
temp := SRC[63:0]
DEST[63:0] := temp
DEST[127:64] := temp
DEST[191:128] := temp
DEST[255:192] := temp
DEST[MAXVL-1:256] := 0
```

VBROADCASTSD (EVEX Encoded Versions)

```
(KL, VL) = (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[63:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VBROADCASTF32x2 (EVEX Encoded Versions)

```
(KL, VL) = (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    n := (j mod 2) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[n+31:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VBROADCASTF128 (VEX.256 Encoded Version)

```
temp := SRC[127:0]
DEST[127:0] := temp
DEST[255:128] := temp
DEST[MAXVL-1:256] := 0
```

VBROADCASTF32X4 (EVEX Encoded Versions)

```
(KL, VL) = (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j* 32
    n := (j modulo 4) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[n+31:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VBROADCASTF64X2 (EVEX Encoded Versions)

```
(KL, VL) = (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    n := (j modulo 2) * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[n+63:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] = 0
            FI
    FI;
ENDFOR;
```

VBROADCASTF32X8 (EVEX.U1.512 Encoded Version)

```
FOR j := 0 TO 15
    i := j * 32
    n := (j modulo 8) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[n+31:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VBROADCASTF64X4 (EVEX.512 Encoded Version)

```
FOR j := 0 TO 7
    i := j * 64
    n := (j modulo 4) * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[n+63:n]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VBROADCASTF32x2 __m512 _mm512_broadcast_f32x2( __m128 a);
VBROADCASTF32x2 __m512 _mm512_mask_broadcast_f32x2(__m512 s, __mmask16 k, __m128 a);
VBROADCASTF32x2 __m512 _mm512_maskz_broadcast_f32x2( __mmask16 k, __m128 a);
VBROADCASTF32x2 __m256 _mm256_broadcast_f32x2( __m128 a);
VBROADCASTF32x2 __m256 _mm256_mask_broadcast_f32x2(__m256 s, __mmask8 k, __m128 a);
VBROADCASTF32x2 __m256 _mm256_maskz_broadcast_f32x2( __mmask8 k, __m128 a);
VBROADCASTF32x4 __m512 _mm512_broadcast_f32x4( __m128 a);
VBROADCASTF32x4 __m512 _mm512_mask_broadcast_f32x4(__m512 s, __mmask16 k, __m128 a);
VBROADCASTF32x4 __m512 _mm512_maskz_broadcast_f32x4( __mmask16 k, __m128 a);
VBROADCASTF32x4 __m256 _mm256_broadcast_f32x4( __m128 a);
VBROADCASTF32x4 __m256 _mm256_mask_broadcast_f32x4(__m256 s, __mmask8 k, __m128 a);
VBROADCASTF32x4 __m256 _mm256_maskz_broadcast_f32x4( __mmask8 k, __m128 a);
VBROADCASTF32x8 __m512 _mm512_broadcast_f32x8( __m256 a);
VBROADCASTF32x8 __m512 _mm512_mask_broadcast_f32x8(__m512 s, __mmask16 k, __m256 a);
VBROADCASTF32x8 __m512 _mm512_maskz_broadcast_f32x8( __mmask16 k, __m256 a);
VBROADCASTF64x2 __m512d _mm512_broadcast_f64x2( __m128d a);
VBROADCASTF64x2 __m512d _mm512_mask_broadcast_f64x2(__m512d s, __mmask8 k, __m128d a);
VBROADCASTF64x2 __m512d _mm512_maskz_broadcast_f64x2( __mmask8 k, __m128d a);
VBROADCASTF64x2 __m256d _mm256_broadcast_f64x2( __m128d a);
VBROADCASTF64x2 __m256d _mm256_mask_broadcast_f64x2(__m256d s, __mmask8 k, __m128d a);
VBROADCASTF64x2 __m256d _mm256_maskz_broadcast_f64x2( __mmask8 k, __m128d a);
VBROADCASTF64x4 __m512d _mm512_broadcast_f64x4( __m256d a);
VBROADCASTF64x4 __m512d _mm512_mask_broadcast_f64x4(__m512d s, __mmask8 k, __m256d a);
VBROADCASTF64x4 __m512d _mm512_maskz_broadcast_f64x4( __mmask8 k, __m256d a);
VBROADCASTSD __m512d _mm512_broadcastsd_pd( __m128d a);
VBROADCASTSD __m512d _mm512_mask_broadcastsd_pd(__m512d s, __mmask8 k, __m128d a);
VBROADCASTSD __m512d _mm512_maskz_broadcastsd_pd(__mmask8 k, __m128d a);
VBROADCASTSD __m256d _mm256_broadcastsd_pd(__m128d a);
VBROADCASTSD __m256d _mm256_mask_broadcastsd_pd(__m256d s, __mmask8 k, __m128d a);
VBROADCASTSD __m256d _mm256_maskz_broadcastsd_pd( __mmask8 k, __m128d a);
VBROADCASTSD __m256d _mm256_broadcast_sd(double *a);
VBROADCASTSS __m512 _mm512_broadcastss_ps( __m128 a);
VBROADCASTSS __m512 _mm512_mask_broadcastss_ps(__m512 s, __mmask16 k, __m128 a);
VBROADCASTSS __m512 _mm512_maskz_broadcastss_ps( __mmask16 k, __m128 a);
VBROADCASTSS __m256 _mm256_broadcastss_ps(__m128 a);
VBROADCASTSS __m256 _mm256_mask_broadcastss_ps(__m256 s, __mmask8 k, __m128 a);
VBROADCASTSS __m256 _mm256_maskz_broadcastss_ps( __mmask8 k, __m128 a);
```

VBROADCASTSS __m128 _mm_broadcastss_ps(__m128 a);
VBROADCASTSS __m128 _mm_mask_broadcastss_ps(__m128 s, __mmask8 k, __m128 a);
VBROADCASTSS __m128 _mm_maskz_broadcastss_ps(__mmask8 k, __m128 a);
VBROADCASTSS __m128 _mm_broadcast_ss(float *a);
VBROADCASTSS __m256 _mm256_broadcast_ss(float *a);
VBROADCASTF128 __m256 _mm256_broadcast_ps(__m128 * a);
VBROADCASTF128 __m256d _mm256_broadcast_pd(__m128d * a);

Exceptions

VEX-encoded instructions, see Table 2-23, "Type 6 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-55, "Type E6 Class Exception Conditions."

Additionally:

#UD	If VEX.L = 0 for VBROADCASTSD or VBROADCASTF128.
	If EVEX.L'L = 0 for VBROADCASTSD/VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF64X2.
	If EVEX.L'L < 10b for VBROADCASTF32X8/VBROADCASTF64X4.

VCMPPH—Compare Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.0F3A.W0 C2 /r /ib VCMPPH k1{k2}, xmm2, xmm3/m128/m16bcst, imm8	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Compare packed FP16 values in xmm3/m128/m16bcst and xmm2 using bits 4:0 of imm8 as a comparison predicate subject to writemask k2, and store the result in mask register k1.
EVEX.256.NP.0F3A.W0 C2 /r /ib VCMPPH k1{k2}, ymm2, ymm3/m256/m16bcst, imm8	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Compare packed FP16 values in ymm3/m256/m16bcst and ymm2 using bits 4:0 of imm8 as a comparison predicate subject to writemask k2, and store the result in mask register k1.
EVEX.512.NP.0F3A.W0 C2 /r /ib VCMPPH k1{k2}, zmm2, zmm3/m512/m16bcst {sae}, imm8	A	V/V	AVX512_FP16 OR AVX10.1	Compare packed FP16 values in zmm3/m512/m16bcst and zmm2 using bits 4:0 of imm8 as a comparison predicate subject to writemask k2, and store the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

This instruction compares packed FP16 values from source operands and stores the result in the destination mask operand. The comparison predicate operand (immediate byte bits 4:0) specifies the type of comparison performed on each of the pairs of packed values. The destination elements are updated according to the writemask.

Operation

```
CASE (imm8 & 0x1F) OF
0: CMP_OPERATOR := EQ_OQ;
1: CMP_OPERATOR := LT_OS;
2: CMP_OPERATOR := LE_OS;
3: CMP_OPERATOR := UNORD_Q;
4: CMP_OPERATOR := NEQ_UQ;
5: CMP_OPERATOR := NLT_US;
6: CMP_OPERATOR := NLE_US;
7: CMP_OPERATOR := ORD_Q;
8: CMP_OPERATOR := EQ_UQ;
9: CMP_OPERATOR := NGE_US;
10: CMP_OPERATOR := NGT_US;
11: CMP_OPERATOR := FALSE_OQ;
12: CMP_OPERATOR := NEQ_OQ;
13: CMP_OPERATOR := GE_OS;
14: CMP_OPERATOR := GT_OS;
15: CMP_OPERATOR := TRUE_UQ;
16: CMP_OPERATOR := EQ_OS;
17: CMP_OPERATOR := LT_OQ;
18: CMP_OPERATOR := LE_OQ;
19: CMP_OPERATOR := UNORD_S;
20: CMP_OPERATOR := NEQ_US;
```

```
21: CMP_OPERATOR := NLT_UQ;
22: CMP_OPERATOR := NLE_UQ;
23: CMP_OPERATOR := ORD_S;
24: CMP_OPERATOR := EQ_US;
25: CMP_OPERATOR := NGE_UQ;
26: CMP_OPERATOR := NGT_UQ;
27: CMP_OPERATOR := FALSE_OS;
28: CMP_OPERATOR := NEQ_OS;
29: CMP_OPERATOR := GE_OQ;
30: CMP_OPERATOR := GT_OQ;
31: CMP_OPERATOR := TRUE_US;
ESAC
```

VCMPPH (EVEX Encoded Versions)

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k2[j] OR *no writemask*:
        IF EVEX.b = 1:
            tsrc2 := SRC2.fp16[0]
        ELSE:
            tsrc2 := SRC2.fp16[j]
        DEST.bit[j] := SRC1.fp16[j] CMP_OPERATOR tsrc2
    ELSE
        DEST.bit[j] := 0

DEST[MAXKL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCMPPH ___mmask8 _mm_cmp_ph_mask (__m128h a, __m128h b, const int imm8);
VCMPPH ___mmask8 _mm_mask_cmp_ph_mask (__mmask8 k1, __m128h a, __m128h b, const int imm8);
VCMPPH ___mmask16 _mm256_cmp_ph_mask (__m256h a, __m256h b, const int imm8);
VCMPPH ___mmask16 _mm256_mask_cmp_ph_mask (__mmask16 k1, __m256h a, __m256h b, const int imm8);
VCMPPH ___mmask32 _mm512_cmp_ph_mask (__m512h a, __m512h b, const int imm8);
VCMPPH ___mmask32 _mm512_mask_cmp_ph_mask (__mmask32 k1, __m512h a, __m512h b, const int imm8);
VCMPPH ___mmask32 _mm512_cmp_round_ph_mask (__m512h a, __m512h b, const int imm8, const int sae);
VCMPPH ___mmask32 _mm512_mask_cmp_round_ph_mask (__mmask32 k1, __m512h a, __m512h b, const int imm8, const int sae);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCMPSH—Compare Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.0F3A.W0 C2 /r /ib VCMPSH k1{k2}, xmm2, xmm3/m16 {sae}, imm8	A	V/V	AVX512_FP16 OR AVX10.1	Compare low FP16 values in xmm3/m16 and xmm2 using bits 4:0 of imm8 as a comparison predicate subject to writemask k2, and store the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

This instruction compares the FP16 values from the lowest element of the source operands and stores the result in the destination mask operand. The comparison predicate operand (immediate byte bits 4:0) specifies the type of comparison performed on the pair of packed FP16 values. The low destination bit is updated according to the write-mask. Bits MAXKL-1:1 of the destination operand are zeroed.

Operation

```
CASE (imm8 & 0x1F) OF
0: CMP_OPERATOR := EQ_OQ;
1: CMP_OPERATOR := LT_OS;
2: CMP_OPERATOR := LE_OS;
3: CMP_OPERATOR := UNORD_Q;
4: CMP_OPERATOR := NEQ_UQ;
5: CMP_OPERATOR := NLT_US;
6: CMP_OPERATOR := NLE_US;
7: CMP_OPERATOR := ORD_Q;
8: CMP_OPERATOR := EQ_UQ;
9: CMP_OPERATOR := NGE_US;
10: CMP_OPERATOR := NGT_US;
11: CMP_OPERATOR := FALSE_OQ;
12: CMP_OPERATOR := NEQ_OQ;
13: CMP_OPERATOR := GE_OS;
14: CMP_OPERATOR := GT_OS;
15: CMP_OPERATOR := TRUE_UQ;
16: CMP_OPERATOR := EQ_OS;
17: CMP_OPERATOR := LT_OQ;
18: CMP_OPERATOR := LE_OQ;
19: CMP_OPERATOR := UNORD_S;
20: CMP_OPERATOR := NEQ_US;
21: CMP_OPERATOR := NLT_UQ;
22: CMP_OPERATOR := NLE_UQ;
23: CMP_OPERATOR := ORD_S;
24: CMP_OPERATOR := EQ_US;
25: CMP_OPERATOR := NGE_UQ;
26: CMP_OPERATOR := NGT_UQ;
27: CMP_OPERATOR := FALSE_OS;
28: CMP_OPERATOR := NEQ_OS;
29: CMP_OPERATOR := GE_OQ;
30: CMP_OPERATOR := GT_OQ;
```

```
31: CMP_OPERATOR := TRUE_US;
ESAC
```

VCMPSH (EVEX Encoded Versions)

```
IF k2[0] OR *no writemask*:
    DEST.bit[0] := SRC1.fp16[0] CMP_OPERATOR SRC2.fp16[0]
ELSE
    DEST.bit[0] := 0

DEST[MAXKL-1:1] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCMPSH __mmask8 _mm_cmp_round_sh_mask (__m128h a, __m128h b, const int imm8, const int sae);
VCMPSH __mmask8 _mm_mask_cmp_round_sh_mask (__mmask8 k1, __m128h a, __m128h b, const int imm8, const int sae);
VCMPSH __mmask8 _mm_cmp_sh_mask (__m128h a, __m128h b, const int imm8);
VCMPSH __mmask8 _mm_mask_cmp_sh_mask (__mmask8 k1, __m128h a, __m128h b, const int imm8);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VCOMISH—Compare Scalar Ordered FP16 Values and Set EFLAGS

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.NP.MAP5.W0 2F /r VCOMISH xmm1, xmm2/m16 {sae}	A	V/V	AVX512_FP16 OR AVX10.1	Compare low FP16 values in xmm1 and xmm2/m16, and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (r)	ModRM:r/m (r)	N/A	N/A

Description

This instruction compares the FP16 values in the low word of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 16-bit memory location.

The VCOMISH instruction differs from the VUCOMISH instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The VUCOMISH instruction signals an invalid numeric exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated. EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCOMISH SRC1, SRC2

```
RESULT := OrderedCompare(SRC1.fp16[0],SRC2.fp16[0])
IF RESULT is UNORDERED:
    ZF, PF, CF := 1, 1, 1
ELSE IF RESULT is GREATER_THAN:
    ZF, PF, CF := 0, 0, 0
ELSE IF RESULT is LESS_THAN:
    ZF, PF, CF := 0, 0, 1
ELSE: // RESULT is EQUALS
    ZF, PF, CF := 1, 0, 0

OF, AF, SF := 0, 0, 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCOMISH int _mm_comi_round_sh (__m128h a, __m128h b, const int imm8, const int sae);
VCOMISH int _mm_comi_sh (__m128h a, __m128h b, const int imm8);
VCOMISH int _mm_comieq_sh (__m128h a, __m128h b);
VCOMISH int _mm_comige_sh (__m128h a, __m128h b);
VCOMISH int _mm_comigt_sh (__m128h a, __m128h b);
VCOMISH int _mm_comile_sh (__m128h a, __m128h b);
VCOMISH int _mm_comilt_sh (__m128h a, __m128h b);
VCOMISH int _mm_comineq_sh (__m128h a, __m128h b);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-50, “Type E3NF Class Exception Conditions.”

VCOMPRESSPD—Store Sparse Packed Double Precision Floating-Point Values Into Dense Memory

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 8A /r VCOMPRESSPD xmm1/m128 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compress packed double precision floating-point values from xmm2 to xmm1/m128 using writemask k1.
EVEX.256.66.0F38.W1 8A /r VCOMPRESSPD ymm1/m256 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compress packed double precision floating-point values from ymm2 to ymm1/m256 using writemask k1.
EVEX.512.66.0F38.W1 8A /r VCOMPRESSPD zmm1/m512 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Compress packed double precision floating-point values from zmm2 using control mask k1 to zmm1/m512.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Compress (store) up to 8 double precision floating-point values from the source operand (the second operand) as a contiguous vector to the destination operand (the first operand) The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 8 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VCOMPRESSPD (EVEX Encoded Versions) Store Form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
SIZE := 64
k := 0
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[k+SIZE-1:k] := SRC[i+63:i]
            k := k + SIZE
    FI;
ENDFOR
```

VCOMPRESSPD (EVEX Encoded Versions) Reg-Reg Form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
SIZE := 64
k := 0
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[k+SIZE-1:k] := SRC[i+63:i]
            k := k + SIZE
    FI;
ENDFOR
IF *merging-masking*
          THEN *DEST[VL-1:k] remains unchanged*
          ELSE DEST[VL-1:k] := 0
FI
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCOMPRESSPD __m512d _mm512_mask_compress_pd(__m512d s, __mmask8 k, __m512d a);
VCOMPRESSPD __m512d _mm512_maskz_compress_pd(__mmask8 k, __m512d a);
VCOMPRESSPD void _mm512_mask_compressstoreu_pd(void * d, __mmask8 k, __m512d a);
VCOMPRESSPD __m256d _mm256_mask_compress_pd(__m256d s, __mmask8 k, __m256d a);
VCOMPRESSPD __m256d _mm256_maskz_compress_pd(__mmask8 k, __m256d a);
VCOMPRESSPD void _mm256_mask_compressstoreu_pd(void * d, __mmask8 k, __m256d a);
VCOMPRESSPD __m128d _mm_mask_compress_pd(__m128d s, __mmask8 k, __m128d a);
VCOMPRESSPD __m128d _mm_maskz_compress_pd(__mmask8 k, __m128d a);
VCOMPRESSPD void _mm_mask_compressstoreu_pd(void * d, __mmask8 k, __m128d a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”
Additionally:

#UD	If EVEX.vvvv != 1111B.

VCOMPRESSPS—Store Sparse Packed Single Precision Floating-Point Values Into Dense Memory

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 8A /r VCOMPRESSPS xmm1/m128 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compress packed single precision floating-point values from xmm2 to xmm1/m128 using writemask k1.
EVEX.256.66.0F38.W0 8A /r VCOMPRESSPS ymm1/m256 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compress packed single precision floating-point values from ymm2 to ymm1/m256 using writemask k1.
EVEX.512.66.0F38.W0 8A /r VCOMPRESSPS zmm1/m512 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Compress packed single precision floating-point values from zmm2 using control mask k1 to zmm1/m512.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Compress (stores) up to 16 single precision floating-point values from the source operand (the second operand) to the destination operand (the first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (a partial vector or possibly non-contiguous if less than 16 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VCOMPRESSPS (EVEX Encoded Versions) Store Form

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
SIZE := 32
k := 0
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            DEST[k+SIZE-1:k] := SRC[i+31:i]
            k := k + SIZE
    FI;
ENDFOR;
```

VCOMPRESSPS (EVEX Encoded Versions) Reg-Reg Form

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
SIZE := 32
k := 0
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            DEST[k+SIZE-1:k] := SRC[i+31:i]
            k := k + SIZE
    FI;
ENDFOR
IF *merging-masking*
    THEN *DEST[VL-1:k] remains unchanged*
    ELSE DEST[VL-1:k] := 0
FI
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCOMPRESSPS __m512 _mm512_mask_compress_ps( __m512 s, __mmask16 k, __m512 a);
VCOMPRESSPS __m512 _mm512_maskz_compress_ps( __mmask16 k, __m512 a);
VCOMPRESSPS void _mm512_mask_compressstoreu_ps( void * d, __mmask16 k, __m512 a);
VCOMPRESSPS __m256 _mm256_mask_compress_ps( __m256 s, __mmask8 k, __m256 a);
VCOMPRESSPS __m256 _mm256_maskz_compress_ps( __mmask8 k, __m256 a);
VCOMPRESSPS void _mm256_mask_compressstoreu_ps( void * d, __mmask8 k, __m256 a);
VCOMPRESSPS __m128 _mm_mask_compress_ps( __m128 s, __mmask8 k, __m128 a);
VCOMPRESSPS __m128 _mm_maskz_compress_ps( __mmask8 k, __m128 a);
VCOMPRESSPS void _mm_mask_compressstoreu_ps( void * d, __mmask8 k, __m128 a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E4.nb. in Table 2-51, "Type E4 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTDQ2PH—Convert Packed Signed Doubleword Integers to Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 5B /r VCVTDQ2PH xmm1{k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert four packed signed doubleword integers from xmm2/m128/m32bcst to four packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 5B /r VCVTDQ2PH xmm1{k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 5B /r VCVTDQ2PH ymm1{k1}{z}, zmm2/m512/m32bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Convert sixteen packed signed doubleword integers from zmm2/m512/m32bcst to sixteen packed FP16 values, and store the result in ymm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts four, eight, or sixteen packed signed doubleword integers in the source operand to four, eight, or sixteen packed FP16 values in the destination operand.

EVEX encoded versions: The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcast from a 32-bit memory location. The destination operand is a YMM/XMM register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

Operation

VCVTDQ2PH DEST, SRC

```
VL = 128, 256 or 512
KL := VL / 32

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.dword[0]
        ELSE
            tsrc := SRC.dword[j]
        DEST.fp16[j] := Convert_integer32_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged
DEST[MAXVL-1:VL/2] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTDQ2PH __m256h _mm512_cvt_roundepi32_ph (__m512i a, int rounding);
VCVTDQ2PH __m256h _mm512_mask_cvt_roundepi32_ph (__m256h src, __mmask16 k, __m512i a, int rounding);
VCVTDQ2PH __m256h _mm512_maskz_cvt_roundepi32_ph (__mmask16 k, __m512i a, int rounding);
VCVTDQ2PH __m128h _mm_cvtepi32_ph (__m128i a);
VCVTDQ2PH __m128h _mm_mask_cvtepi32_ph (__m128h src, __mmask8 k, __m128i a);
VCVTDQ2PH __m128h _mm_maskz_cvtepi32_ph (__mmask8 k, __m128i a);
VCVTDQ2PH __m128h _mm256_cvtepi32_ph (__m256i a);
VCVTDQ2PH __m128h _mm256_mask_cvtepi32_ph (__m128h src, __mmask8 k, __m256i a);
VCVTDQ2PH __m128h _mm256_maskz_cvtepi32_ph (__mmask8 k, __m256i a);
VCVTDQ2PH __m256h _mm512_cvtepi32_ph (__m512i a);
VCVTDQ2PH __m256h _mm512_mask_cvtepi32_ph (__m256h src, __mmask16 k, __m512i a);
VCVTDQ2PH __m256h _mm512_maskz_cvtepi32_ph (__mmask16 k, __m512i a);

SIMD Floating-Point Exceptions

Overflow, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTNE2PS2BF16—Convert Two Packed Single Data to One Packed BF16 Data

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.0F38.W0 72 /r VCVTNE2PS2BF16 xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	(AVX512_BF16 AND AVX512VL) OR AVX10.1	Convert packed single data from xmm2 and xmm3/m128/m32bcst to packed BF16 data in xmm1 with writemask k1.
EVEX.256.F2.0F38.W0 72 /r VCVTNE2PS2BF16 ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	(AVX512_BF16 AND AVX512VL) OR AVX10.1	Convert packed single data from ymm2 and ymm3/m256/m32bcst to packed BF16 data in ymm1 with writemask k1.
EVEX.512.F2.0F38.W0 72 /r VCVTNE2PS2BF16 zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	(AVX512_BF16 AND AVX512F) OR AVX10.1	Convert packed single data from zmm2 and zmm3/m512/m32bcst to packed BF16 data in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Converts two SIMD registers of packed single data into a single register of packed BF16 data.

This instruction does not support memory fault suppression.

This instruction uses "Round to nearest (even)" rounding mode. Output denormals are always flushed to zero and input denormals are always treated as zero. MXCSR is not consulted nor updated. No floating-point exceptions are generated.

Operation

VCVTNE2PS2BF16 dest, src1, src2

```
VL = (128, 256, 512)
KL = VL/16

origdest := dest
FOR i := 0 to KL-1:
    IF k1[ i ] or *no writemask*:
        IF i < KL/2:
            IF src2 is memory and evex.b == 1:
                t := src2.fp32[0]
            ELSE:
                t := src2.fp32[ i ]
        ELSE:
            t := src1.fp32[ i-KL/2]

        // See VCVTNEPS2BF16 for definition of convert helper function
        dest.word[i] := convert_fp32_to_bfloat16(t)

    ELSE IF *zeroing*:
        dest.word[ i ] := 0
    ELSE:  // Merge masking, dest element unchanged
        dest.word[ i ] := origdest.word[ i ]
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTNE2PS2BF16 __m128bh _mm_cvtne2ps_pbh (__m128, __m128);
VCVTNE2PS2BF16 __m128bh _mm_mask_cvtne2ps_pbh (__m128bh, __mmask8, __m128, __m128);
VCVTNE2PS2BF16 __m128bh _mm_maskz_cvtne2ps_pbh (__mmask8, __m128, __m128);
VCVTNE2PS2BF16 __m256bh _mm256_cvtne2ps_pbh (__m256, __m256);
VCVTNE2PS2BF16 __m256bh _mm256_mask_cvtne2ps_pbh (__m256bh, __mmask16, __m256, __m256);
VCVTNE2PS2BF16 __m256bh _mm256_maskz_cvtne2ps_ pbh (__mmask16, __m256, __m256);
VCVTNE2PS2BF16 __m512bh _mm512_cvtne2ps_pbh (__m512, __m512);
VCVTNE2PS2BF16 __m512bh _mm512_mask_cvtne2ps_pbh (__m512bh, __mmask32, __m512, __m512);
VCVTNE2PS2BF16 __m512bh _mm512_maskz_cvtne2ps_pbh (__mmask32, __m512, __m512);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-52, "Type E4NF Class Exception Conditions."

VCVTNEEBF162PS—Convert Even Elements of Packed BF16 Values to FP32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.F3.0F38.W0 B0 !(11):rrr:bbb VCVTNEEBF162PS xmm1, m128	A	V/V	AVX_NE_ CONVERT	Convert even elements of packed BF16 values from m128 to FP32 values and store in xmm1.
VEX.256.F3.0F38.W0 B0 !(11):rrr:bbb VCVTNEEBF162PS ymm1, m256	A	V/V	AVX_NE_ CONVERT	Convert even elements of packed BF16 values from m256 to FP32 values and store in ymm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction loads packed BF16 elements from memory, converts the even elements to FP32, and writes the result to the destination SIMD register.

This instruction does not generate floating-point exceptions and does not consult or update MXCSR.

Since any BF16 number can be represented in FP32, the conversion result is exact and no rounding is needed.

Operation

VCVTNEEBF162PS dest, src (VEX encoded version)

```
VL = (128, 256)
KL = VL/32

FOR i in range(0, KL):
    dest.dword[i] = make_fp32(src.dword[i].word[0])
DEST[MAXVL-1:VL] := 0
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VCVTNEEBF162PS __m128 _mm_cvtneebf16_ps (const __m128bh* __A);
VCVTNEEBF162PS __m256 _mm256_cvtneebf16_ps (const __m256bh* __A);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions.”

VCVTNEEPH2PS—Convert Even Elements of Packed FP16 Values to FP32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 B0 !(11):rrr:bbb VCVTNEEPH2PS xmm1, m128	A	V/V	AVX_NE_ CONVERT	Convert even elements of packed FP16 values from m128 to FP32 values and store in xmm1.
VEX.256.66.0F38.W0 B0 !(11):rrr:bbb VCVTNEEPH2PS ymm1, m256	A	V/V	AVX_NE_ CONVERT	Convert even elements of packed FP16 values from m256 to FP32 values and store in ymm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction loads packed FP16 elements from memory, converts the even elements to FP32, and writes the result to the destination SIMD register.

This instruction does not generate floating-point exceptions and does not consult or update MXCSR.

Input FP16 denormals are converted to normal FP32 numbers and not treated as zero. Since any FP16 number can be represented in FP32, the conversion result is exact and no rounding is needed.

Operation

VCVTNEEPH2PS dest, src (VEX encoded version)

```
VL = (128, 256)
KL = VL/32

FOR i in range(0, KL):
    dest.dword[i] = convert_fp16_to_fp32(src.dword[i].word[0]) //SAE
DEST[MAXVL-1:VL] := 0
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VCVTNEEPH2PS __m128 _mm_cvtneeph_ps (const __m128h* __A);
VCVTNEEPH2PS __m256 _mm256_cvtneeph_ps (const __m256h* __A);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions."

VCVTNEOBF162PS—Convert Odd Elements of Packed BF16 Values to FP32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.F2.0F38.W0 B0 !(11):rrr:bbb VCVTNEOBF162PS xmm1, m128	A	V/V	AVX_NE_ CONVERT	Convert odd elements of packed BF16 values from m128 to FP32 values and store in xmm1.
VEX.256.F2.0F38.W0 B0 !(11):rrr:bbb VCVTNEOBF162PS ymm1, m256	A	V/V	AVX_NE_ CONVERT	Convert odd elements of packed BF16 values from m256 to FP32 values and store in ymm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction loads packed BF16 elements from memory, converts the odd elements to FP32, and writes the result to the destination SIMD register.

This instruction does not generate floating-point exceptions and does not consult or update MXCSR.

Since any BF16 number can be represented in FP32, the conversion result is exact and no rounding is needed.

Operation

VCVTNEOBF162PS dest, src (VEX encoded version)

```
VL = (128, 256)
KL = VL/32

FOR i in range(0, KL):
    dest.dword[i] = make_fp32(src.dword[i].word[1])
DEST[MAXVL-1:VL] := 0
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VCVTNEOBF162PS __m128 _mm_cvtneobf16_ps (const __m128bh* __A);
VCVTNEOBF162PS __m256 _mm256_cvtneobf16_ps (const __m256bh* __A);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions.”

VCVTNEOPH2PS—Convert Odd Elements of Packed FP16 Values to FP32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.NP.0F38.W0 B0 !(11):rrr:bbb VCVTNEOPH2PS xmm1, m128	A	V/V	AVX_NE_ CONVERT	Convert odd elements of packed FP16 values from m128 to FP32 values and store in xmm1.
VEX.256.NP.0F38.W0 B0 !(11):rrr:bbb VCVTNEOPH2PS ymm1, m256	A	V/V	AVX_NE_ CONVERT	Convert odd elements of packed FP16 values from m256 to FP32 values and store in ymm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction loads packed FP16 elements from memory, converts the odd elements to FP32, and writes the result to the destination SIMD register.

This instruction does not generate floating-point exceptions and does not consult or update MXCSR.

Input FP16 denormals are converted to normal FP32 numbers and not treated as zero. Since any FP16 number can be represented in FP32, the conversion result is exact and no rounding is needed.

Operation

VCVTNEOPH2PS dest, src (VEX encoded version)

```
VL = (128, 256)
KL = VL/32

FOR i in range(0, KL):
    dest.dword[i] = convert_fp16_to_fp32(src.dword[i].word[1]) //SAE
DEST[MAXVL-1:VL] := 0
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VCVTNEOPH2PS __m128 _mm_cvtneoph_ps (const __m128h* __A);
VCVTNEOPH2PS __m256 _mm256_cvtneoph_ps (const __m256h* __A);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions."

VCVTNEPS2BF16—Convert Packed Single Data to Packed BF16 Data

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.F3.0F38.W0 72 /r VCVTNEPS2BF16 xmm1, xmm2/m128	A	V/V	AVX_NE_ CONVERT	Convert packed single precision floating-point values from xmm2/m128 to packed BF16 values and store in xmm1.
VEX.256.F3.0F38.W0 72 /r VCVTNEPS2BF16 xmm1, ymm2/m256	A	V/V	AVX_NE_ CONVERT	Convert packed single precision floating-point values from ymm2/m256 to packed BF16 values and store in xmm1.
EVEX.128.F3.0F38.W0 72 /r VCVTNEPS2BF16 xmm1{k1}{z}, xmm2/m128/m32bcst	B	V/V	(AVX512_BF16 AND AVX512VL) OR AVX10.1	Convert packed single data from xmm2/m128 to packed BF16 data in xmm1 with writemask k1.
EVEX.256.F3.0F38.W0 72 /r VCVTNEPS2BF16 xmm1{k1}{z}, ymm2/m256/m32bcst	B	V/V	(AVX512_BF16 AND AVX512VL) OR AVX10.1	Convert packed single data from ymm2/m256 to packed BF16 data in xmm1 with writemask k1.
EVEX.512.F3.0F38.W0 72 /r VCVTNEPS2BF16 ymm1{k1}{z}, zmm2/m512/m32bcst	B	V/V	(AVX512_BF16 AND AVX512F) OR AVX10.1	Convert packed single data from zmm2/m512 to packed BF16 data in ymm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction loads packed FP32 elements from a SIMD register or memory, converts the elements to BF16, and writes the result to the destination SIMD register.

The upper bits of the destination register beyond the down-converted BF16 elements are zeroed.

This instruction uses "Round to nearest (even)" rounding mode. Output denormals are always flushed to zero and input denormals are always treated as zero. MXCSR is not consulted nor updated.

As the instruction operand encoding table shows, the EVEX.vvvv field is not used for encoding an operand. EVEX.vvvv is reserved and must be 0b1111 otherwise instructions will #UD.

Operation

```
Define convert_fp32_to_bfloat16(x):
    IF x is zero or denormal:
        dest[15] := x[31] // sign preserving zero (denormal go to zero)
        dest[14:0] := 0
    ELSE IF x is infinity:
        dest[15:0] := x[31:16]
    ELSE IF x is NAN:
        dest[15:0] := x[31:16] // truncate and set MSB of the mantissa to force QNAN
        dest[6] := 1
    ELSE // normal number
        LSB := x[16]
        rounding_bias := 0x00007FFF + LSB
        temp[31:0] := x[31:0] + rounding_bias // integer add
        dest[15:0] := temp[31:16]
    RETURN dest
```

VCVTNEPS2BF16 dest, src (VEX encoded version)

```
VL = (128, 256)
KL = VL/16

FOR i := 0 to KL/2-1:
    t := src.fp32[i]
    dest.word[i] := convert_fp32_to_bfloat16(t)

DEST[MAXVL-1:VL/2] := 0
```

VCVTNEPS2BF16 dest, src (EVEX encoded version)

```
VL = (128, 256, 512)
KL = VL/16

origdest := dest
FOR i := 0 to KL/2-1:
    IF k1[ i ] or *no writemask*:
        IF src is memory and evex.b == 1:
            t := src.fp32[0]
        ELSE:
            t := src.fp32[ i ]

        dest.word[i] := convert_fp32_to_bfloat16(t)

    ELSE IF *zeroing*:
        dest.word[ i ] := 0
    ELSE:  // Merge masking, dest element unchanged
        dest.word[ i ] := origdest.word[ i ]
DEST[MAXVL-1:VL/2] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTNEPS2BF16 __m128bh _mm_cvtneps_avx_pbh (__m128 __A);
VCVTNEPS2BF16 __m128bh _mm256_cvtneps_avx_pbh (__m256 __A);
VCVTNEPS2BF16 __m128bh _mm_cvtneps_pbh (__m128 a);
VCVTNEPS2BF16 __m128bh _mm_cvtneps_pbh (__m128 __A);
VCVTNEPS2BF16 __m128bh _mm_mask_cvtneps_pbh (__m128bh src, __mmask8 k, __m128 a);
VCVTNEPS2BF16 __m128bh _mm_maskz_cvtneps_pbh (__mmask8 k, __m128 a);
VCVTNEPS2BF16 __m128bh _mm256_cvtneps_pbh (__m256 a);
VCVTNEPS2BF16 __m128bh _mm256_cvtneps_pbh (__m256 __A);
VCVTNEPS2BF16 __m128bh _mm256_mask_cvtneps_pbh (__m128bh src, __mmask8 k, __m256 a);
VCVTNEPS2BF16 __m128bh _mm256_maskz_cvtneps_pbh (__mmask8 k, __m256 a);
VCVTNEPS2BF16 __m256bh _mm512_cvtneps_pbh (__m512 a);
VCVTNEPS2BF16 __m256bh _mm512_mask_cvtneps_pbh (__m256bh src, __mmask16 k, __m512 a);
VCVTNEPS2BF16 __m256bh _mm512_maskz_cvtneps_pbh (__mmask16 k, __m512 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-51, “Type E4 Class Exception Conditions.”

VCVTPD2PH—Convert Packed Double Precision FP Values to Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.W1 5A /r VCVTPD2PH xmm1{k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert two packed double precision floating-point values in xmm2/m128/m64bcst to two packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP5.W1 5A /r VCVTPD2PH xmm1{k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.512.66.MAP5.W1 5A /r VCVTPD2PH xmm1{k1}{z}, zmm2/m512/m64bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight packed FP16 values, and store the result in ymm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts two, four, or eight packed double precision floating-point values in the source operand (second operand) to two, four, or eight packed FP16 values in the destination operand (first operand). When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasts from a 64-bit memory location. The destination operand is a XMM register conditionally updated with writemask k1. The upper bits (MAXVL-1:128/64/32) of the corresponding destination are zeroed.

EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

This instruction uses MXCSR.DAZ for handling FP64 inputs. FP16 outputs can be normal or denormal, and are not conditionally flushed to zero.

Operation

VCVTPD2PH DEST, SRC

```
VL = 128, 256 or 512
KL := VL / 64

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.double[0]
        ELSE
            tsrc := SRC.double[j]
        DEST.fp16[j] := Convert_fp64_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL/4] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPD2PH __m128h _mm512_cvt_roundpd_ph (__m512d a, int rounding);
VCVTPD2PH __m128h _mm512_mask_cvt_roundpd_ph (__m128h src, __mmask8 k, __m512d a, int rounding);
VCVTPD2PH __m128h _mm512_maskz_cvt_roundpd_ph (__mmask8 k, __m512d a, int rounding);
VCVTPD2PH __m128h _mm_cvtpd_ph (__m128d a);
VCVTPD2PH __m128h _mm_mask_cvtpd_ph (__m128h src, __mmask8 k, __m128d a);
VCVTPD2PH __m128h _mm_maskz_cvtpd_ph (__mmask8 k, __m128d a);
VCVTPD2PH __m128h _mm256_cvtpd_ph (__m256d a);
VCVTPD2PH __m128h _mm256_mask_cvtpd_ph (__m128h src, __mmask8 k, __m256d a);
VCVTPD2PH __m128h _mm256_maskz_cvtpd_ph (__mmask8 k, __m256d a);
VCVTPD2PH __m128h _mm512_cvtpd_ph (__m512d a);
VCVTPD2PH __m128h _mm512_mask_cvtpd_ph (__m128h src, __mmask8 k, __m512d a);
VCVTPD2PH __m128h _mm512_maskz_cvtpd_ph (__mmask8 k, __m512d a);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTPD2QQ—Convert Packed Double Precision Floating-Point Values to Packed Quadword Integers

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W1 7B /r VCVTPD2QQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert two packed double precision floating-point values from xmm2/m128/m64bcst to two packed signed quadword integers in xmm1 with writemask k1.
EVEX.256.66.0F.W1 7B /r VCVTPD2QQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert four packed double precision floating-point values from ymm2/m256/m64bcst to four packed signed quadword integers in ymm1 with writemask k1.
EVEX.512.66.0F.W1 7B /r VCVTPD2QQ zmm1 {k1}{z}, zmm2/m512/m64bcst {er}	A	V/V	AVX512DQ OR AVX10.1	Convert eight packed double precision floating-point values from zmm2/m512/m64bcst to eight packed signed quadword integers in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts packed double precision floating-point values in the source operand (second operand) to packed quad-word integers in the destination operand (first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000_00000000H is returned.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTPD2QQ (EVEX Encoded Version) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL == 512) AND (EVEX.b == 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_Double_Precision_Floating_Point_To_QuadInteger(SRC[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
```

```
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTPD2QQ (EVEX Encoded Version) When SRC Operand is a Memory Source
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] :=          Convert_Double_Precision_Floating_Point_To_QuadInteger(SRC[63:0])
                ELSE
                    DEST[i+63:i] := Convert_Double_Precision_Floating_Point_To_QuadInteger(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPD2QQ __m512i _mm512_cvtpd_epi64(__m512d a);
VCVTPD2QQ __m512i _mm512_mask_cvtpd_epi64(__m512i s, __mmask8 k, __m512d a);
VCVTPD2QQ __m512i _mm512_maskz_cvtpd_epi64(__mmask8 k, __m512d a);
VCVTPD2QQ __m512i _mm512_cvt_roundpd_epi64(__m512d a, int r);
VCVTPD2QQ __m512i _mm512_mask_cvt_roundpd_epi64(__m512i s, __mmask8 k, __m512d a, int r);
VCVTPD2QQ __m512i _mm512_maskz_cvt_roundpd_epi64(__mmask8 k, __m512d a, int r);
VCVTPD2QQ __m256i _mm256_mask_cvtpd_epi64(__m256i s, __mmask8 k, __m256d a);
VCVTPD2QQ __m256i _mm256_maskz_cvtpd_epi64(__mmask8 k, __m256d a);
VCVTPD2QQ __m128i _mm_mask_cvtpd_epi64(__m128i s, __mmask8 k, __m128d a);
VCVTPD2QQ __m128i _mm_maskz_cvtpd_epi64(__mmask8 k, __m128d a);
VCVTPD2QQ __m256i _mm256_cvtpd_epi64 (__m256d src)
VCVTPD2QQ __m128i _mm_cvtpd_epi64 (__m128d src)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."
Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTPD2UDQ—Convert Packed Double Precision Floating-Point Values to Packed Unsigned Doubleword Integers

Opcode Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.0F.W1 79 /r VCVTPD2UDQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert two packed double precision floating-point values in xmm2/m128/m64bcst to two unsigned doubleword integers in xmm1 subject to writemask k1.
EVEX.256.0F.W1 79 /r VCVTPD2UDQ xmm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four unsigned doubleword integers in xmm1 subject to writemask k1.
EVEX.512.0F.W1 79 /r VCVTPD2UDQ ymm1 {k1}{z}, zmm2/m512/m64bcst {er}	A	V/V	AVX512F OR AVX10.1	Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight unsigned doubleword integers in ymm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts packed double precision floating-point values in the source operand (the second operand) to packed unsigned doubleword integers in the destination operand (the first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFFH is returned.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. The upper bits (MAXVL-1:256) of the corresponding destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTPD2UDQ (EVEX Encoded Versions) When SRC2 Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_UInteger(SRC[k+63:k])
```

```
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

VCVTPD2UDQ (EVEX Encoded Versions) When SRC Operand is a Memory Source
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_UInteger(SRC[63:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_UInteger(SRC[k+63:k])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPD2UDQ __m256i _mm512_cvtpd_epu32(__m512d a);
VCVTPD2UDQ __m256i _mm512_mask_cvtpd_epu32(__m256i s, __mmask8 k, __m512d a);
VCVTPD2UDQ __m256i _mm512_maskz_cvtpd_epu32(__mmask8 k, __m512d a);
VCVTPD2UDQ __m256i _mm512_cvt_roundpd_epu32(__m512d a, int r);
VCVTPD2UDQ __m256i _mm512_mask_cvt_roundpd_epu32(__m256i s, __mmask8 k, __m512d a, int r);
VCVTPD2UDQ __m256i _mm512_maskz_cvt_roundpd_epu32(__mmask8 k, __m512d a, int r);
VCVTPD2UDQ __m128i _mm256_mask_cvtpd_epu32(__m128i s, __mmask8 k, __m256d a);
VCVTPD2UDQ __m128i _mm256_maskz_cvtpd_epu32(__mmask8 k, __m256d a);
VCVTPD2UDQ __m128i _mm_mask_cvtpd_epu32(__m128i s, __mmask8 k, __m128d a);
VCVTPD2UDQ __m128i _mm_maskz_cvtpd_epu32(__mmask8 k, __m128d a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTPD2UQQ—Convert Packed Double Precision Floating-Point Values to Packed Unsigned Quadword Integers

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W1 79 /r VCVTPD2UQQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert two packed double precision floating-point values from xmm2/mem to two packed unsigned quadword integers in xmm1 with writemask k1.
EVEX.256.66.0F.W1 79 /r VCVTPD2UQQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert fourth packed double precision floating-point values from ymm2/mem to four packed unsigned quadword integers in ymm1 with writemask k1.
EVEX.512.66.0F.W1 79 /r VCVTPD2UQQ zmm1 {k1}{z}, zmm2/m512/m64bcst {er}	A	V/V	AVX512DQ OR AVX10.1	Convert eight packed double precision floating-point values from zmm2/mem to eight packed unsigned quadword integers in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts packed double precision floating-point values in the source operand (second operand) to packed unsigned quadword integers in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFF_FFFFFFFFH is returned.

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTPD2UQQ (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL == 512) AND (EVEX.b == 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_Double_Precision_Floating_Point_To_UQuadInteger(SRC[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
```

```
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTPD2UQQ (EVEX Encoded Versions) When SRC Operand is a Memory Source
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] :=
            Convert_Double_Precision_Floating_Point_To_UQuadInteger(SRC[63:0])
                ELSE
                    DEST[i+63:i] :=
            Convert_Double_Precision_Floating_Point_To_UQuadInteger(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPD2UQQ __m512i _mm512_cvtpd_epu64(__m512d a);
VCVTPD2UQQ __m512i _mm512_mask_cvtpd_epu64(__m512i s, __mmask8 k, __m512d a);
VCVTPD2UQQ __m512i _mm512_maskz_cvtpd_epu64(__mmask8 k, __m512d a);
VCVTPD2UQQ __m512i _mm512_cvt_roundpd_epu64(__m512d a, int r);
VCVTPD2UQQ __m512i _mm512_mask_cvt_roundpd_epu64(__m512i s, __mmask8 k, __m512d a, int r);
VCVTPD2UQQ __m512i _mm512_maskz_cvt_roundpd_epu64(__mmask8 k, __m512d a, int r);
VCVTPD2UQQ __m256i _mm256_mask_cvtpd_epu64(__m256i s, __mmask8 k, __m256d a);
VCVTPD2UQQ __m256i _mm256_maskz_cvtpd_epu64(__mmask8 k, __m256d a);
VCVTPD2UQQ __m128i _mm_mask_cvtpd_epu64(__m128i s, __mmask8 k, __m128d a);
VCVTPD2UQQ __m128i _mm_maskz_cvtpd_epu64(__mmask8 k, __m128d a);
VCVTPD2UQQ __m256i _mm256_cvtpd_epu64 (__m256d src)
VCVTPD2UQQ __m128i _mm_cvtpd_epu64 (__m128d src)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”
Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTPH2DQ—Convert Packed FP16 Values to Signed Doubleword Integers

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.W0 5B /r VCVTPH2DQ xmm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert four packed FP16 values in xmm2/m64/m16bcst to four signed doubleword integers, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP5.W0 5B /r VCVTPH2DQ ymm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight signed doubleword integers, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP5.W0 5B /r VCVTPH2DQ zmm1{k1}{z}, ymm2/m256/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Convert sixteen packed FP16 values in ymm2/m256/m16bcst to sixteen signed doubleword integers, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Half	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed FP16 values in the source operand to signed doubleword integers in destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

The destination elements are updated according to the writemask.

Operation

VCVTPH2DQ DEST, SRC

```
VL = 128, 256 or 512
KL := VL / 32

IF *SRC is a register* and (VL = 512) and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.dword[j] := Convert_fp16_to_integer32(tsrc)
    ELSE IF *zeroing*:
        DEST.dword[j] := 0
    // else dest.dword[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPH2DQ __m512i _mm512_cvt_roundph_epi32 (__m256h a, int rounding);
VCVTPH2DQ __m512i _mm512_mask_cvt_roundph_epi32 (__m512i src, __mmask16 k, __m256h a, int rounding);
VCVTPH2DQ __m512i _mm512_maskz_cvt_roundph_epi32 (__mmask16 k, __m256h a, int rounding);
VCVTPH2DQ __m128i _mm_cvtph_epi32 (__m128h a);
VCVTPH2DQ __m128i _mm_mask_cvtph_epi32 (__m128i src, __mmask8 k, __m128h a);
VCVTPH2DQ __m128i _mm_maskz_cvtph_epi32 (__mmask8 k, __m128h a);
VCVTPH2DQ __m256i _mm256_cvtph_epi32 (__m128h a);
VCVTPH2DQ __m256i _mm256_mask_cvtph_epi32 (__m256i src, __mmask8 k, __m128h a);
VCVTPH2DQ __m256i _mm256_maskz_cvtph_epi32 (__mmask8 k, __m128h a);
VCVTPH2DQ __m512i _mm512_cvtph_epi32 (__m256h a);
VCVTPH2DQ __m512i _mm512_mask_cvtph_epi32 (__m512i src, __mmask16 k, __m256h a);
VCVTPH2DQ __m512i _mm512_maskz_cvtph_epi32 (__mmask16 k, __m256h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTPH2PD—Convert Packed FP16 Values to FP64 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 5A /r VCVTPH2PD xmm1{k1}{z}, xmm2/m32/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert packed FP16 values in xmm2/m32/m16bcst to FP64 values, and store result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 5A /r VCVTPH2PD ymm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert packed FP16 values in xmm2/m64/m16bcst to FP64 values, and store result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 5A /r VCVTPH2PD zmm1{k1}{z}, xmm2/m128/m16bcst {sae}	A	V/V	AVX512_FP16 OR AVX10.1	Convert packed FP16 values in xmm2/m128/m16bcst to FP64 values, and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Quarter	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed FP16 values to FP64 values in the destination register. The destination elements are updated according to the writemask.

This instruction handles both normal and denormal FP16 inputs.

Operation

VCVTPH2PD DEST, SRC

```
VL = 128, 256, or 512
KL := VL/64

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.fp64[j] := Convert_fp16_to_fp64(tsrc)
    ELSE IF *zeroing*:
        DEST.fp64[j] := 0
    // else dest.fp64[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPH2PD __m512d _mm512_cvt_roundph_pd (__m128h a, int sae);
VCVTPH2PD __m512d _mm512_mask_cvt_roundph_pd (__m512d src, __mmask8 k, __m128h a, int sae);
VCVTPH2PD __m512d _mm512_maskz_cvt_roundph_pd (__mmask8 k, __m128h a, int sae);
VCVTPH2PD __m128d _mm_cvtph_pd (__m128h a);
VCVTPH2PD __m128d _mm_mask_cvtph_pd (__m128d src, __mmask8 k, __m128h a);
VCVTPH2PD __m128d _mm_maskz_cvtph_pd (__mmask8 k, __m128h a);
VCVTPH2PD __m256d _mm256_cvtph_pd (__m128h a);
VCVTPH2PD __m256d _mm256_mask_cvtph_pd (__m256d src, __mmask8 k, __m128h a);
VCVTPH2PD __m256d _mm256_maskz_cvtph_pd (__mmask8 k, __m128h a);
VCVTPH2PD __m512d _mm512_cvtph_pd (__m128h a);
VCVTPH2PD __m512d _mm512_mask_cvtph_pd (__m512d src, __mmask8 k, __m128h a);
VCVTPH2PD __m512d _mm512_maskz_cvtph_pd (__mmask8 k, __m128h a);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

VCVTPH2PS/VCVTPH2PSX—Convert Packed FP16 Values to Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 13 /r VCVTPH2PS xmm1, xmm2/m64	A	V/V	F16C	Convert four packed FP16 values in xmm2/m64 to packed single precision floating-point value in xmm1.
VEX.256.66.0F38.W0 13 /r VCVTPH2PS ymm1, xmm2/m128	A	V/V	F16C	Convert eight packed FP16 values in xmm2/m128 to packed single precision floating-point value in ymm1.
EVEX.128.66.0F38.W0 13 /r VCVTPH2PS xmm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed FP16 values in xmm2/m64 to packed single precision floating-point values in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 13 /r VCVTPH2PS ymm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert eight packed FP16 values in xmm2/m128 to packed single precision floating-point values in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 13 /r VCVTPH2PS zmm1 {k1}{z}, ymm2/m256 {sae}	B	V/V	AVX512F OR AVX10.1	Convert sixteen packed FP16 values in ymm2/m256 to packed single precision floating-point values in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 13 /r VCVTPH2PSX xmm1{k1}{z}, xmm2/m64/m16bcst	C	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert four packed FP16 values in xmm2/m64/m16bcst to four packed single precision floating-point values, and store result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 13 /r VCVTPH2PSX ymm1{k1}{z}, xmm2/m128/m16bcst	C	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight packed single precision floating-point values, and store result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 13 /r VCVTPH2PSX zmm1{k1}{z}, ymm2/m256/m16bcst {sae}	C	V/V	AVX512_FP16 OR AVX10.1	Convert sixteen packed FP16 values in ymm2/m256/m16bcst to sixteen packed single precision floating-point values, and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Half Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
C	Half	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed half precision (16-bits) floating-point values in the low-order bits of the source operand (the second operand) to packed single precision floating-point values and writes the converted values into the destination operand (the first operand).

If case of a denormal operand, the correct normal result is returned. MXCSR.DAZ is ignored and is treated as if it 0. No denormal exception is reported on MXCSR.

VEX.128 version: The source operand is a XMM register or 64-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 version: The source operand is a XMM register or 128-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64-bits) register or a 256/128/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

The diagram below illustrates how data is converted from four packed half precision (in 64 bits) to four single precision (in 128 bits) floating-point values.

Note: VEX.vvvv and EVEX.vvvv are reserved (must be 1111b).

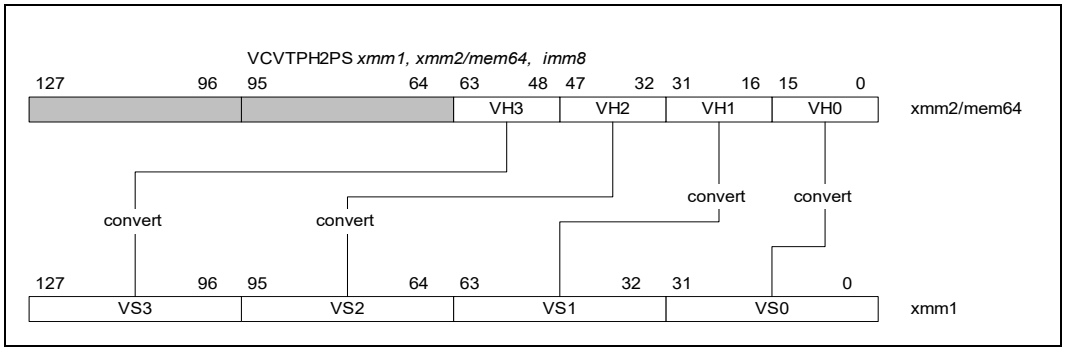


Figure 5-6. VCVTPH2PS (128-bit Version)

The VCVTPH2PSX instruction is a new form of the PH to PS conversion instruction, encoded in map 6. The previous version of the instruction, VCVTPH2PS, that is present in AVX512F (encoded in map 2, 0F38) does not support embedded broadcasting. The VCVTPH2PSX instruction has the embedded broadcasting option available.

The instructions associated with AVX512_FP16 always handle FP16 denormal number inputs; denormal inputs are not treated as zero.

Operation

```
vCvt_h2s(SRC1[15:0])
{
RETURN Cvt_Half_Precision_To_Single_Precision(SRC1[15:0]);
}
```

VCVTPH2PS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            vCvt_h2s(SRC[k+15:k])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTPH2PS (VEX.256 Encoded Version)

```
DEST[31:0] := vCvt_h2s(SRC1[15:0]);
DEST[63:32] := vCvt_h2s(SRC1[31:16]);
DEST[95:64] := vCvt_h2s(SRC1[47:32]);
DEST[127:96] := vCvt_h2s(SRC1[63:48]);
DEST[159:128] := vCvt_h2s(SRC1[79:64]);
DEST[191:160] := vCvt_h2s(SRC1[95:80]);
DEST[223:192] := vCvt_h2s(SRC1[111:96]);
DEST[255:224] := vCvt_h2s(SRC1[127:112]);
DEST[MAXVL-1:256] := 0
```

VCVTPH2PS (VEX.128 Encoded Version)

```
DEST[31:0] := vCvt_h2s(SRC1[15:0]);
DEST[63:32] := vCvt_h2s(SRC1[31:16]);
DEST[95:64] := vCvt_h2s(SRC1[47:32]);
DEST[127:96] := vCvt_h2s(SRC1[63:48]);
DEST[MAXVL-1:128] := 0
```

VCVTPH2PSX DEST, SRC

```
VL = 128, 256, or 512
KL := VL/32

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.fp32[j] := Convert_fp16_to_fp32(tsrc)
    ELSE IF *zeroing*:
        DEST.fp32[j] := 0
    // else dest.fp32[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPH2PS __m512 _mm512_cvtph_ps(__m256i a);
VCVTPH2PS __m512 _mm512_mask_cvtph_ps(__m512 s, __mmask16 k, __m256i a);
VCVTPH2PS __m512 _mm512_maskz_cvtph_ps(__mmask16 k, __m256i a);
VCVTPH2PS __m512 _mm512_cvt_roundph_ps(__m256i a, int sae);
VCVTPH2PS __m512 _mm512_mask_cvt_roundph_ps(__m512 s, __mmask16 k, __m256i a, int sae);
VCVTPH2PS __m512 _mm512_maskz_cvt_roundph_ps(__mmask16 k, __m256i a, int sae);
VCVTPH2PS __m256 _mm256_mask_cvtph_ps(__m256 s, __mmask8 k, __m128i a);
VCVTPH2PS __m256 _mm256_maskz_cvtph_ps(__mmask8 k, __m128i a);
VCVTPH2PS __m128 _mm_mask_cvtph_ps(__m128 s, __mmask8 k, __m128i a);
VCVTPH2PS __m128 _mm_maskz_cvtph_ps(__mmask8 k, __m128i a);
VCVTPH2PS __m128 _mm_cvtph_ps (__m128i m1);
VCVTPH2PS __m256 _mm256_cvtph_ps (__m128i m1)

VCVTPH2PSX __m512 _mm512_cvtx_roundph_ps (__m256h a, int sae);
VCVTPH2PSX __m512 _mm512_mask_cvtx_roundph_ps (__m512 src, __mmask16 k, __m256h a, int sae);
VCVTPH2PSX __m512 _mm512_maskz_cvtx_roundph_ps (__mmask16 k, __m256h a, int sae);
VCVTPH2PSX __m128 _mm_cvtxph_ps (__m128h a);
VCVTPH2PSX __m128 _mm_mask_cvtxph_ps (__m128 src, __mmask8 k, __m128h a);
VCVTPH2PSX __m128 _mm_maskz_cvtxph_ps (__mmask8 k, __m128h a);
VCVTPH2PSX __m256 _mm256_cvtxph_ps (__m128h a);
VCVTPH2PSX __m256 _mm256_mask_cvtxph_ps (__m256 src, __mmask8 k, __m128h a);
VCVTPH2PSX __m256 _mm256_maskz_cvtxph_ps (__mmask8 k, __m128h a);
VCVTPH2PSX __m512 _mm512_cvtxph_ps (__m256h a);
VCVTPH2PSX __m512 _mm512_mask_cvtxph_ps (__m512 src, __mmask16 k, __m256h a);
VCVTPH2PSX __m512 _mm512_maskz_cvtxph_ps (__mmask16 k, __m256h a);

SIMD Floating-Point Exceptions

VEX-encoded instructions: Invalid.

EVEX-encoded instructions: Invalid.
EVEX-encoded instructions with broadcast (VCVTPH2PSX): Invalid, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-26, "Type 11 Class Exception Conditions" (do not report #AC).

EVEX-encoded instructions, see Table 2-62, "Type E11 Class Exception Conditions."

EVEX-encoded instructions with broadcast (VCVTPH2PSX), see Table 2-46, "Type E2 Class Exception Conditions."

Additionally:

#UD	If VEX.W=1.
#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

VCVTPH2QQ—Convert Packed FP16 Values to Signed Quadword Integer Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.W0 7B /r VCVTPH2QQ xmm1{k1}{z}, xmm2/m32/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert two packed FP16 values in xmm2/m32/m16bcst to two signed quadword integers, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP5.W0 7B /r VCVTPH2QQ ymm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert four packed FP16 values in xmm2/m64/m16bcst to four signed quadword integers, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP5.W0 7B /r VCVTPH2QQ zmm1{k1}{z}, xmm2/m128/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight signed quadword integers, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Quarter	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed FP16 values in the source operand to signed quadword integers in destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000_00000000H is returned.

The destination elements are updated according to the writemask.

Operation

VCVTPH2QQ DEST, SRC

```
VL = 128, 256 or 512
KL := VL / 64

IF *SRC is a register* and (VL = 512) and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.qword[j] := Convert_fp16_to_integer64(tsrc)
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    // else dest.qword[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPH2QQ __m512i _mm512_cvt_roundph_epi64 (__m128h a, int rounding);
VCVTPH2QQ __m512i _mm512_mask_cvt_roundph_epi64 (__m512i src, __mmask8 k, __m128h a, int rounding);
VCVTPH2QQ __m512i _mm512_maskz_cvt_roundph_epi64 (__mmask8 k, __m128h a, int rounding);
VCVTPH2QQ __m128i _mm_cvtph_epi64 (__m128h a);
VCVTPH2QQ __m128i _mm_mask_cvtph_epi64 (__m128i src, __mmask8 k, __m128h a);
VCVTPH2QQ __m128i _mm_maskz_cvtph_epi64 (__mmask8 k, __m128h a);
VCVTPH2QQ __m256i _mm256_cvtph_epi64 (__m128h a);
VCVTPH2QQ __m256i _mm256_mask_cvtph_epi64 (__m256i src, __mmask8 k, __m128h a);
VCVTPH2QQ __m256i _mm256_maskz_cvtph_epi64 (__mmask8 k, __m128h a);
VCVTPH2QQ __m512i _mm512_cvtph_epi64 (__m128h a);
VCVTPH2QQ __m512i _mm512_mask_cvtph_epi64 (__m512i src, __mmask8 k, __m128h a);
VCVTPH2QQ __m512i _mm512_maskz_cvtph_epi64 (__mmask8 k, __m128h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTPH2UDQ—Convert Packed FP16 Values to Unsigned Doubleword Integers

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 79 /r VCVTPH2UDQ xmm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert four packed FP16 values in xmm2/m64/m16bcst to four unsigned doubleword integers, and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 79 /r VCVTPH2UDQ ymm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight unsigned doubleword integers, and store the result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 79 /r VCVTPH2UDQ zmm1{k1}{z}, ymm2/m256/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Convert sixteen packed FP16 values in ymm2/m256/m16bcst to sixteen unsigned doubleword integers, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Half	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed FP16 values in the source operand to unsigned doubleword integers in destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFFH is returned.

The destination elements are updated according to the writemask.

Operation

VCVTPH2UDQ DEST, SRC

```
VL = 128, 256 or 512
KL := VL / 32

IF *SRC is a register* and (VL = 512) and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
            DEST.dword[j] := Convert_fp16_to_unsigned_integer32(tsrc)
    ELSE IF *zeroing*:
        DEST.dword[j] := 0
    // else dest.dword[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPH2UDQ __m512i _mm512_cvt_roundph_epu32 (__m256h a, int rounding);
VCVTPH2UDQ __m512i _mm512_mask_cvt_roundph_epu32 (__m512i src, __mmask16 k, __m256h a, int rounding);
VCVTPH2UDQ __m512i _mm512_maskz_cvt_roundph_epu32 (__mmask16 k, __m256h a, int rounding);
VCVTPH2UDQ __m128i _mm_cvtph_epu32 (__m128h a);
VCVTPH2UDQ __m128i _mm_mask_cvtph_epu32 (__m128i src, __mmask8 k, __m128h a);
VCVTPH2UDQ __m128i _mm_maskz_cvtph_epu32 (__mmask8 k, __m128h a);
VCVTPH2UDQ __m256i _mm256_cvtph_epu32 (__m128h a);
VCVTPH2UDQ __m256i _mm256_mask_cvtph_epu32 (__m256i src, __mmask8 k, __m128h a);
VCVTPH2UDQ __m256i _mm256_maskz_cvtph_epu32 (__mmask8 k, __m128h a);
VCVTPH2UDQ __m512i _mm512_cvtph_epu32 (__m256h a);
VCVTPH2UDQ __m512i _mm512_mask_cvtph_epu32 (__m512i src, __mmask16 k, __m256h a);
VCVTPH2UDQ __m512i _mm512_maskz_cvtph_epu32 (__mmask16 k, __m256h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTPH2UQQ—Convert Packed FP16 Values to Unsigned Quadword Integers

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.W0 79 /r VCVTPH2UQQ xmm1{k1}{z}, xmm2/m32/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert two packed FP16 values in xmm2/m32/m16bcst to two unsigned quadword integers, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP5.W0 79 /r VCVTPH2UQQ ymm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert four packed FP16 values in xmm2/m64/m16bcst to four unsigned quadword integers, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP5.W0 79 /r VCVTPH2UQQ zmm1{k1}{z}, xmm2/m128/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight unsigned quadword integers, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Quarter	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed FP16 values in the source operand to unsigned quadword integers in destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFF_FFFFFFFFH is returned.

The destination elements are updated according to the writemask.

Operation

VCVTPH2UQQ DEST, SRC

```
VL = 128, 256 or 512
KL := VL / 64

IF *SRC is a register* and (VL = 512) and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.qword[j] := Convert_fp16_to_unsigned_integer64(tsrc)
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    // else dest.qword[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPH2UQQ __m512i _mm512_cvt_roundph_epu64 (__m128h a, int rounding);
VCVTPH2UQQ __m512i _mm512_mask_cvt_roundph_epu64 (__m512i src, __mmask8 k, __m128h a, int rounding);
VCVTPH2UQQ __m512i _mm512_maskz_cvt_roundph_epu64 (__mmask8 k, __m128h a, int rounding);
VCVTPH2UQQ __m128i _mm_cvtph_epu64 (__m128h a);
VCVTPH2UQQ __m128i _mm_mask_cvtph_epu64 (__m128i src, __mmask8 k, __m128h a);
VCVTPH2UQQ __m128i _mm_maskz_cvtph_epu64 (__mmask8 k, __m128h a);
VCVTPH2UQQ __m256i _mm256_cvtph_epu64 (__m128h a);
VCVTPH2UQQ __m256i _mm256_mask_cvtph_epu64 (__m256i src, __mmask8 k, __m128h a);
VCVTPH2UQQ __m256i _mm256_maskz_cvtph_epu64 (__mmask8 k, __m128h a);
VCVTPH2UQQ __m512i _mm512_cvtph_epu64 (__m128h a);
VCVTPH2UQQ __m512i _mm512_mask_cvtph_epu64 (__m512i src, __mmask8 k, __m128h a);
VCVTPH2UQQ __m512i _mm512_maskz_cvtph_epu64 (__mmask8 k, __m128h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTPH2UW—Convert Packed FP16 Values to Unsigned Word Integers

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 7D /r VCVTPH2UW xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert packed FP16 values in xmm2/m128/m16bcst to unsigned word integers, and store the result in xmm1.
EVEX.256.NP.MAP5.W0 7D /r VCVTPH2UW ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert packed FP16 values in ymm2/m256/m16bcst to unsigned word integers, and store the result in ymm1.
EVEX.512.NP.MAP5.W0 7D /r VCVTPH2UW zmm1{k1}{z}, zmm2/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Convert packed FP16 values in zmm2/m512/m16bcst to unsigned word integers, and store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed FP16 values in the source operand to unsigned word integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFH is returned.

The destination elements are updated according to the writemask.

Operation

VCVTPH2UW DEST, SRC

```
VL = 128, 256 or 512
KL := VL / 16

IF *SRC is a register* and (VL = 512) and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.word[j] := Convert_fp16_to_unsigned_integer16(tsrc)
    ELSE IF *zeroing*:
        DEST.word[j] := 0
    // else dest.word[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPH2UW __m512i _mm512_cvt_roundph_epu16 (__m512h a, int sae);
VCVTPH2UW __m512i _mm512_mask_cvt_roundph_epu16 (__m512i src, __mmask32 k, __m512h a, int sae);
VCVTPH2UW __m512i _mm512_maskz_cvt_roundph_epu16 (__mmask32 k, __m512h a, int sae);
VCVTPH2UW __m128i _mm_cvtph_epu16 (__m128h a);
VCVTPH2UW __m128i _mm_mask_cvtph_epu16 (__m128i src, __mmask8 k, __m128h a);
VCVTPH2UW __m128i _mm_maskz_cvtph_epu16 (__mmask8 k, __m128h a);
VCVTPH2UW __m256i _mm256_cvtph_epu16 (__m256h a);
VCVTPH2UW __m256i _mm256_mask_cvtph_epu16 (__m256i src, __mmask16 k, __m256h a);
VCVTPH2UW __m256i _mm256_maskz_cvtph_epu16 (__mmask16 k, __m256h a);
VCVTPH2UW __m512i _mm512_cvtph_epu16 (__m512h a);
VCVTPH2UW __m512i _mm512_mask_cvtph_epu16 (__m512i src, __mmask32 k, __m512h a);
VCVTPH2UW __m512i _mm512_maskz_cvtph_epu16 (__mmask32 k, __m512h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

VCVTPH2W—Convert Packed FP16 Values to Signed Word Integers

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.W0 7D /r VCVTPH2W xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert packed FP16 values in xmm2/m128/m16bcst to signed word integers, and store the result in xmm1.
EVEX.256.66.MAP5.W0 7D /r VCVTPH2W ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert packed FP16 values in ymm2/m256/m16bcst to signed word integers, and store the result in ymm1.
EVEX.512.66.MAP5.W0 7D /r VCVTPH2W zmm1{k1}{z}, zmm2/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Convert packed FP16 values in zmm2/m512/m16bcst to signed word integers, and store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed FP16 values in the source operand to signed word integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 8000H is returned.

The destination elements are updated according to the writemask.

Operation

VCVTPH2W DEST, SRC

```
VL = 128, 256 or 512
KL := VL / 16

IF *SRC is a register* and (VL = 512) and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.word[j] := Convert_fp16_to_integer16(tsrc)
    ELSE IF *zeroing*:
        DEST.word[j] := 0
    // else dest.word[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPH2W __m512i _mm512_cvt_roundph_epi16 (__m512h a, int rounding);
VCVTPH2W __m512i _mm512_mask_cvt_roundph_epi16 (__m512i src, __mmask32 k, __m512h a, int rounding);
VCVTPH2W __m512i _mm512_maskz_cvt_roundph_epi16 (__mmask32 k, __m512h a, int rounding);
VCVTPH2W __m128i _mm_cvtph_epi16 (__m128h a);
VCVTPH2W __m128i _mm_mask_cvtph_epi16 (__m128i src, __mmask8 k, __m128h a);
VCVTPH2W __m128i _mm_maskz_cvtph_epi16 (__mmask8 k, __m128h a);
VCVTPH2W __m256i _mm256_cvtph_epi16 (__m256h a);
VCVTPH2W __m256i _mm256_mask_cvtph_epi16 (__m256i src, __mmask16 k, __m256h a);
VCVTPH2W __m256i _mm256_maskz_cvtph_epi16 (__mmask16 k, __m256h a);
VCVTPH2W __m512i _mm512_cvtph_epi16 (__m512h a);
VCVTPH2W __m512i _mm512_mask_cvtph_epi16 (__m512i src, __mmask32 k, __m512h a);
VCVTPH2W __m512i _mm512_maskz_cvtph_epi16 (__mmask32 k, __m512h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

VCVTPS2PH—Convert Single Precision FP Value to 16-bit FP Value

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F3A.W0 1D /r ib VCVTPS2PH xmm1/m64, xmm2, imm8	A	V/V	F16C	Convert four packed single precision floating-point values in xmm2 to packed half-precision (16-bit) floating-point values in xmm1/m64. Imm8 provides rounding controls.
VEX.256.66.0F3A.W0 1D /r ib VCVTPS2PH xmm1/m128, ymm2, imm8	A	V/V	F16C	Convert eight packed single precision floating-point values in ymm2 to packed half-precision (16-bit) floating-point values in xmm1/m128. Imm8 provides rounding controls.
EVEX.128.66.0F3A.W0 1D /r ib VCVTPS2PH xmm1/m64 {k1}{z}, xmm2, imm8	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed single-precision floating-point values in xmm2 to packed half-precision (16-bit) floating-point values in xmm1/m64. Imm8 provides rounding controls.
EVEX.256.66.0F3A.W0 1D /r ib VCVTPS2PH xmm1/m128 {k1}{z}, ymm2, imm8	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert eight packed single-precision floating-point values in ymm2 to packed half-precision (16-bit) floating-point values in xmm1/m128. Imm8 provides rounding controls.
EVEX.512.66.0F3A.W0 1D /r ib VCVTPS2PH ymm1/m256 {k1}{z}, zmm2 {sae}, imm8	B	V/V	AVX512F OR AVX10.1	Convert sixteen packed single-precision floating-point values in zmm2 to packed half-precision (16-bit) floating-point values in ymm1/m256. Imm8 provides rounding controls.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
B	Half Mem	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A

Description

Convert packed single precision floating values in the source operand to half-precision (16-bit) floating-point values and store to the destination operand. The rounding mode is specified using the immediate field (imm8).

Underflow results (i.e., tiny results) are converted to denormals. MXCSR.FTZ is ignored. If a source element is denormal relative to the input format with DM masked and at least one of PM or UM unmasked; a SIMD exception will be raised with DE, UE and PE set.

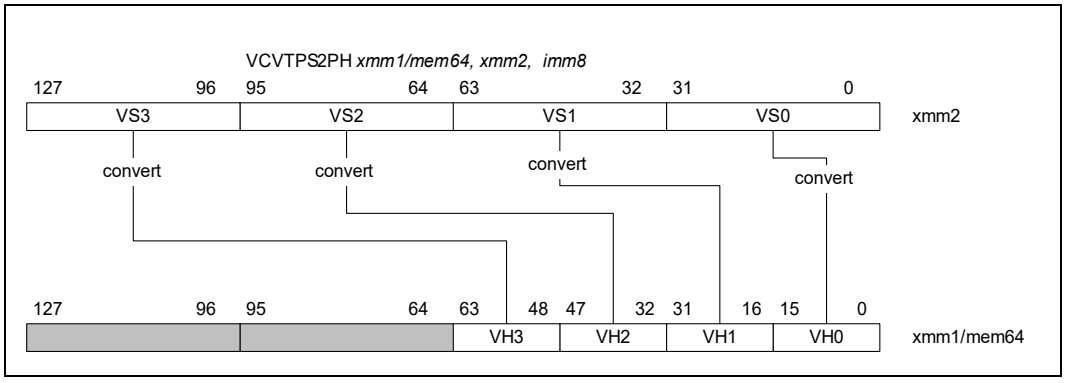


Figure 5-7. VCVTPS2PH (128-bit Version)

The immediate byte defines several bit fields that control rounding operation. The effect and encoding of the RC field are listed in Table 5-3.

Table 5-3. Immediate Byte Encoding for 16-bit Floating-Point Conversion Instructions

Bits	Field Name/value	Description	Comment
Imm[1:0]	RC=00B	Round to nearest even	If Imm[2] = 0
	RC=01B	Round down	
	RC=10B	Round up	
	RC=11B	Truncate	
Imm[2]	MS1=0	Use imm[1:0] for rounding	Ignore MXCSR.RC
	MS1=1	Use MXCSR.RC for rounding	
Imm[7:3]	Ignored	Ignored by processor	

VEX.128 version: The source operand is a XMM register. The destination operand is a XMM register or 64-bit memory location. If the destination operand is a register then the upper bits (MAXVL-1:64) of corresponding register are zeroed.

VEX.256 version: The source operand is a YMM register. The destination operand is a XMM register or 128-bit memory location. If the destination operand is a register, the upper bits (MAXVL-1:128) of the corresponding destination register are zeroed.

Note: VEX.vvvv and EVEX.vvvv are reserved (must be 1111b).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM (low 64-bits) register or a 256/128/64-bit memory location, conditionally updated with writemask k1. Bits (MAXVL-1:256/128/64) of the corresponding destination register are zeroed.

Operation

```
vCvt_s2h(SRC1[31:0])
{
IF Imm[2] = 0
THEN    ; using Imm[1:0] for rounding control, see Table 5-3
    RETURN Cvt_Single_Precision_To_Half_Precision_FP_Imm(SRC1[31:0]);
ELSE    ; using MXCSR.RC for rounding control
    RETURN Cvt_Single_Precision_To_Half_Precision_FP_Mxcsr(SRC1[31:0]);
FI;
}
```

VCVTPS2PH (EVEX Encoded Versions) When DEST is a Register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 16
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] :=
            vCvt_s2h(SRC[k+31:k])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

VCVTPS2PH (EVEX Encoded Versions) When DEST is Memory

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 16
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] :=
            vCvt_s2h(SRC[k+31:k])
        ELSE
            *DEST[i+15:i] remains unchanged*    ; merging-masking
    FI;
ENDFOR
```

VCVTPS2PH (VEX.256 Encoded Version)

```
DEST[15:0] := vCvt_s2h(SRC1[31:0]);
DEST[31:16] := vCvt_s2h(SRC1[63:32]);
DEST[47:32] := vCvt_s2h(SRC1[95:64]);
DEST[63:48] := vCvt_s2h(SRC1[127:96]);
DEST[79:64] := vCvt_s2h(SRC1[159:128]);
DEST[95:80] := vCvt_s2h(SRC1[191:160]);
DEST[111:96] := vCvt_s2h(SRC1[223:192]);
DEST[127:112] := vCvt_s2h(SRC1[255:224]);
DEST[MAXVL-1:128] := 0
```

VCVTPS2PH (VEX.128 Encoded Version)

```
DEST[15:0] := vCvt_s2h(SRC1[31:0]);
DEST[31:16] := vCvt_s2h(SRC1[63:32]);
DEST[47:32] := vCvt_s2h(SRC1[95:64]);
DEST[63:48] := vCvt_s2h(SRC1[127:96]);
DEST[MAXVL-1:64] := 0
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPS2PH __m256i _mm512_cvtps_ph(__m512 a);
VCVTPS2PH __m256i _mm512_mask_cvtps_ph(__m256i s, __mmask16 k,__m512 a);
VCVTPS2PH __m256i _mm512_maskz_cvtps_ph(__mmask16 k,__m512 a);
VCVTPS2PH __m256i _mm512_cvt_roundps_ph(__m512 a, const int imm);
VCVTPS2PH __m256i _mm512_mask_cvt_roundps_ph(__m256i s, __mmask16 k,__m512 a, const int imm);
VCVTPS2PH __m256i _mm512_maskz_cvt_roundps_ph(__mmask16 k,__m512 a, const int imm);
VCVTPS2PH __m128i _mm256_mask_cvtps_ph(__m128i s, __mmask8 k,__m256 a);
VCVTPS2PH __m128i _mm256_maskz_cvtps_ph(__mmask8 k,__m256 a);
VCVTPS2PH __m128i _mm_mask_cvtps_ph(__m128i s, __mmask8 k,__m128 a);
VCVTPS2PH __m128i _mm_maskz_cvtps_ph(__mmask8 k,__m128 a);
VCVTPS2PH __m128i _mm_cvtps_ph (__m128 m1, const int imm);
VCVTPS2PH __m128i _mm256_cvtps_ph(__m256 m1, const int imm);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal (if MXCSR.DAZ=0).

Other Exceptions

VEX-encoded instructions, see Table 2-26, “Type 11 Class Exception Conditions” (do not report #AC);

EVEX-encoded instructions, see Table 2-62, “Type E11 Class Exception Conditions.”

Additionally:

#UD	If VEX.W=1.
#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

VCVTPS2PHX—Convert Packed Single Precision Floating-Point Values to Packed FP16 Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.W0 1D /r VCVTPS2PHX xmm1{k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert four packed single precision floating-point values in xmm2/m128/m32bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP5.W0 1D /r VCVTPS2PHX xmm1{k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert eight packed single precision floating-point values in ymm2/m256/m32bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.512.66.MAP5.W0 1D /r VCVTPS2PHX ymm1{k1}{z}, zmm2/m512/m32bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Convert sixteen packed single precision floating-point values in zmm2 /m512/m32bcst to packed FP16 values, and store the result in ymm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed single precision floating values in the source operand to FP16 values and stores to the destination operand.

The VCVTPS2PHX instruction supports broadcasting.

This instruction uses MXCSR.DAZ for handling FP32 inputs. FP16 outputs can be normal or denormal numbers, and are not conditionally flushed based on MXCSR settings.

Operation

VCVTPS2PHX DEST, SRC (AVX512_FP16 Load Version With Broadcast Support)

```
VL = 128, 256, or 512
KL := VL / 32

IF *SRC is a register* and (VL == 512) and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp32[0]
        ELSE
            tsrc := SRC.fp32[j]
        DEST.fp16[j] := Convert_fp32_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL/2] := 0
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPS2PHX __m256h _mm512_cvtx_roundps_ph (__m512 a, int rounding);
VCVTPS2PHX __m256h _mm512_mask_cvtx_roundps_ph (__m256h src, __mmask16 k, __m512 a, int rounding);
VCVTPS2PHX __m256h _mm512_maskz_cvtx_roundps_ph (__mmask16 k, __m512 a, int rounding);
VCVTPS2PHX __m128h _mm_cvtxps_ph (__m128 a);
VCVTPS2PHX __m128h _mm_mask_cvtxps_ph (__m128h src, __mmask8 k, __m128 a);
VCVTPS2PHX __m128h _mm_maskz_cvtxps_ph (__mmask8 k, __m128 a);
VCVTPS2PHX __m128h _mm256_cvtxps_ph (__m256 a);
VCVTPS2PHX __m128h _mm256_mask_cvtxps_ph (__m128h src, __mmask8 k, __m256 a);
VCVTPS2PHX __m128h _mm256_maskz_cvtxps_ph (__mmask8 k, __m256 a);
VCVTPS2PHX __m256h _mm512_cvtxps_ph (__m512 a);
VCVTPS2PHX __m256h _mm512_mask_cvtxps_ph (__m256h src, __mmask16 k, __m512 a);
VCVTPS2PHX __m256h _mm512_maskz_cvtxps_ph (__mmask16 k, __m512 a);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal (if MXCSR.DAZ=0).

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If VEX.W=1.
#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

VCVTPS2QQ—Convert Packed Single Precision Floating-Point Values to Packed Signed Quadword Integer Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 7B /r VCVTPS2QQ xmm1 {k1}{z}, xmm2/m64/m32bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed signed quadword values in xmm1 subject to writemask k1.
EVEX.256.66.0F.W0 7B /r VCVTPS2QQ ymm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed quadword values in ymm1 subject to writemask k1.
EVEX.512.66.0F.W0 7B /r VCVTPS2QQ zmm1 {k1}{z}, ymm2/m256/m32bcst {er}	A	V/V	AVX512DQ OR AVX10.1	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed quadword values in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts eight packed single precision floating-point values in the source operand to eight signed quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000_00000000H is returned.

The source operand is a YMM/XMM/XMM (low 64- bits) register or a 256/128/64-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTPS2QQ (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL == 512) AND (EVEX.b == 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_Single_Precision_To_QuadInteger(SRC[k+31:k])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
```

```
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTPS2QQ (EVEX Encoded Versions) When SRC Operand is a Memory Source
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] :=
            Convert_Single_Precision_To_QuadInteger(SRC[31:0])
                ELSE
                    DEST[i+63:i] :=
            Convert_Single_Precision_To_QuadInteger(SRC[k+31:k])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPS2QQ __m512i _mm512_cvtps_epi64(__m512 a);
VCVTPS2QQ __m512i _mm512_mask_cvtps_epi64(__m512i s, __mmask16 k, __m512 a);
VCVTPS2QQ __m512i _mm512_maskz_cvtps_epi64(__mmask16 k, __m512 a);
VCVTPS2QQ __m512i _mm512_cvt_roundps_epi64(__m512 a, int r);
VCVTPS2QQ __m512i _mm512_mask_cvt_roundps_epi64(__m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2QQ __m512i _mm512_maskz_cvt_roundps_epi64(__mmask16 k, __m512 a, int r);
VCVTPS2QQ __m256i _mm256_cvtps_epi64(__m256 a);
VCVTPS2QQ __m256i _mm256_mask_cvtps_epi64(__m256i s, __mmask8 k, __m256 a);
VCVTPS2QQ __m256i _mm256_maskz_cvtps_epi64(__mmask8 k, __m256 a);
VCVTPS2QQ __m128i _mm_cvtps_epi64(__m128 a);
VCVTPS2QQ __m128i _mm_mask_cvtps_epi64(__m128i s, __mmask8 k, __m128 a);
VCVTPS2QQ __m128i _mm_maskz_cvtps_epi64(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."
Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTPS2UDQ—Convert Packed Single Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.0F.W0 79 /r VCVTPS2UDQ xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned doubleword values in xmm1 subject to writemask k1.
EVEX.256.0F.W0 79 /r VCVTPS2UDQ ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned doubleword values in ymm1 subject to writemask k1.
EVEX.512.0F.W0 79 /r VCVTPS2UDQ zmm1 {k1}{z}, zmm2/m512/m32bcst {er}	A	V/V	AVX512F OR AVX10.1	Convert sixteen packed single precision floating-point values from zmm2/m512/m32bcst to sixteen packed unsigned doubleword values in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts sixteen packed single precision floating-point values in the source operand to sixteen unsigned double-word integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFFH is returned.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTPS2UDQ (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_UInteger(SRC[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTPS2UDQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no *
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_UInteger(SRC[31:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_UInteger(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPS2UDQ __m512i _mm512_cvtps_epu32(__m512 a);
VCVTPS2UDQ __m512i _mm512_mask_cvtps_epu32(__m512i s, __mmask16 k, __m512 a);
VCVTPS2UDQ __m512i _mm512_maskz_cvtps_epu32(__mmask16 k, __m512 a);
VCVTPS2UDQ __m512i _mm512_cvt_roundps_epu32(__m512 a, int r);
VCVTPS2UDQ __m512i _mm512_mask_cvt_roundps_epu32(__m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2UDQ __m512i _mm512_maskz_cvt_roundps_epu32(__mmask16 k, __m512 a, int r);
VCVTPS2UDQ __m256i _mm256_cvtps_epu32(__m256d a);
VCVTPS2UDQ __m256i _mm256_mask_cvtps_epu32(__m256i s, __mmask8 k, __m256 a);
VCVTPS2UDQ __m256i _mm256_maskz_cvtps_epu32(__mmask8 k, __m256 a);
VCVTPS2UDQ __m128i _mm_cvtps_epu32(__m128 a);
VCVTPS2UDQ __m128i _mm_mask_cvtps_epu32(__m128i s, __mmask8 k, __m128 a);
VCVTPS2UDQ __m128i _mm_maskz_cvtps_epu32(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTPS2UQQ—Convert Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 79 /r VCVTPS2UQQ xmm1 {k1}{z}, xmm2/m64/m32bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert two packed single precision floating-point values from zmm2/m64/m32bcst to two packed unsigned quadword values in zmm1 subject to writemask k1.
EVEX.256.66.0F.W0 79 /r VCVTPS2UQQ ymm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned quadword values in ymm1 subject to writemask k1.
EVEX.512.66.0F.W0 79 /r VCVTPS2UQQ zmm1 {k1}{z}, ymm2/m256/m32bcst {er}	A	V/V	AVX512DQ OR AVX10.1	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned quadword values in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts up to eight packed single precision floating-point values in the source operand to unsigned quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFF_FFFFFFFFH is returned.

The source operand is a YMM/XMM/XMM (low 64- bits) register or a 256/128/64-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTPS2UQQ (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL == 512) AND (EVEX.b == 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_Single_Precision_To_UQuadInteger(SRC[k+31:k])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
```

```
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTPS2UQQ (EVEX Encoded Versions) When SRC Operand is a Memory Source
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] :=
            Convert_Single_Precision_To_UQuadInteger(SRC[31:0])
                ELSE
                    DEST[i+63:i] :=
            Convert_Single_Precision_To_UQuadInteger(SRC[k+31:k])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTPS2UQQ __m512i _mm512_cvtps_epu64( __m512 a);
VCVTPS2UQQ __m512i _mm512_mask_cvtps_epu64( __m512i s, __mmask16 k, __m512 a);
VCVTPS2UQQ __m512i _mm512_maskz_cvtps_epu64( __mmask16 k, __m512 a);
VCVTPS2UQQ __m512i _mm512_cvt_roundps_epu64( __m512 a, int r);
VCVTPS2UQQ __m512i _mm512_mask_cvt_roundps_epu64( __m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2UQQ __m512i _mm512_maskz_cvt_roundps_epu64( __mmask16 k, __m512 a, int r);
VCVTPS2UQQ __m256i _mm256_cvtps_epu64( __m256 a);
VCVTPS2UQQ __m256i _mm256_mask_cvtps_epu64( __m256i s, __mmask8 k, __m256 a);
VCVTPS2UQQ __m256i _mm256_maskz_cvtps_epu64( __mmask8 k, __m256 a);
VCVTPS2UQQ __m128i _mm_cvtps_epu64( __m128 a);
VCVTPS2UQQ __m128i _mm_mask_cvtps_epu64( __m128i s, __mmask8 k, __m128 a);
VCVTPS2UQQ __m128i _mm_maskz_cvtps_epu64( __mmask8 k, __m128 a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”
Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTQQ2PD—Convert Packed Quadword Integers to Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F.W1 E6 /r VCVTQQ2PD xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert two packed quadword integers from xmm2/m128/m64bcst to packed double precision floating-point values in xmm1 with writemask k1.
EVEX.256.F3.0F.W1 E6 /r VCVTQQ2PD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert four packed quadword integers from ymm2/m256/m64bcst to packed double precision floating-point values in ymm1 with writemask k1.
EVEX.512.F3.0F.W1 E6 /r VCVTQQ2PD zmm1 {k1}{z}, zmm2/m512/m64bcst {er}	A	V/V	AVX512DQ OR AVX10.1	Convert eight packed quadword integers from zmm2/m512/m64bcst to eight packed double precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts packed quadword integers in the source operand (second operand) to packed double precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTQQ2PD (EVEX2 Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL == 512) AND (EVEX.b == 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_QuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTQQ2PD (EVEX Encoded Versions) when SRC Operand is a Memory Source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] :=
            Convert_QuadInteger_To_Double_Precision_Floating_Point(SRC[63:0])
                ELSE
                    DEST[i+63:i] :=
            Convert_QuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTQQ2PD __m512d _mm512_cvtepi64_pd( __m512i a);
VCVTQQ2PD __m512d _mm512_mask_cvtepi64_pd( __m512d s, __mmask16 k, __m512i a);
VCVTQQ2PD __m512d _mm512_maskz_cvtepi64_pd( __mmask16 k, __m512i a);
VCVTQQ2PD __m512d _mm512_cvt_roundepi64_pd( __m512i a, int r);
VCVTQQ2PD __m512d _mm512_mask_cvt_roundepi64_pd( __m512d s, __mmask8 k, __m512i a, int r);
VCVTQQ2PD __m512d _mm512_maskz_cvt_roundepi64_pd( __mmask8 k, __m512i a, int r);
VCVTQQ2PD __m256d _mm256_mask_cvtepi64_pd( __m256d s, __mmask8 k, __m256i a);
VCVTQQ2PD __m256d _mm256_maskz_cvtepi64_pd( __mmask8 k, __m256i a);
VCVTQQ2PD __m128d _mm_mask_cvtepi64_pd( __m128d s, __mmask8 k, __m128i a);
VCVTQQ2PD __m128d _mm_maskz_cvtepi64_pd( __mmask8 k, __m128i a);
```

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTQQ2PH—Convert Packed Signed Quadword Integers to Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W1 5B /r VCVTQQ2PH xmm1{k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert two packed signed quadword integers in xmm2/m128/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W1 5B /r VCVTQQ2PH xmm1{k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert four packed signed quadword integers in ymm2/m256/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.512.NP.MAP5.W1 5B /r VCVTQQ2PH xmm1{k1}{z}, zmm2/m512/m64bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Convert eight packed signed quadword integers in zmm2/m512/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed signed quadword integers in the source operand to packed FP16 values in the destination operand. The destination elements are updated according to the writemask.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

Operation

VCVTQQ2PH DEST, SRC

```
VL = 128, 256 or 512
KL := VL / 64

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.qword[0]
        ELSE
            tsrc := SRC.qword[j]
        DEST.fp16[j] := Convert_integer64_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL/4] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTQQ2PH __m128h _mm512_cvt_roundepi64_ph (__m512i a, int rounding);
VCVTQQ2PH __m128h _mm512_mask_cvt_roundepi64_ph (__m128h src, __mmask8 k, __m512i a, int rounding);
VCVTQQ2PH __m128h _mm512_maskz_cvt_roundepi64_ph (__mmask8 k, __m512i a, int rounding);
VCVTQQ2PH __m128h _mm_cvtepi64_ph (__m128i a);
VCVTQQ2PH __m128h _mm_mask_cvtepi64_ph (__m128h src, __mmask8 k, __m128i a);
VCVTQQ2PH __m128h _mm_maskz_cvtepi64_ph (__mmask8 k, __m128i a);
VCVTQQ2PH __m128h _mm256_cvtepi64_ph (__m256i a);
VCVTQQ2PH __m128h _mm256_mask_cvtepi64_ph (__m128h src, __mmask8 k, __m256i a);
VCVTQQ2PH __m128h _mm256_maskz_cvtepi64_ph (__mmask8 k, __m256i a);
VCVTQQ2PH __m128h _mm512_cvtepi64_ph (__m512i a);
VCVTQQ2PH __m128h _mm512_mask_cvtepi64_ph (__m128h src, __mmask8 k, __m512i a);
VCVTQQ2PH __m128h _mm512_maskz_cvtepi64_ph (__mmask8 k, __m512i a);

SIMD Floating-Point Exceptions

Overflow, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTQQ2PS—Convert Packed Quadword Integers to Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.0F.W1 5B /r VCVTQQ2PS xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert two packed quadword integers from xmm2/mem to packed single precision floating-point values in xmm1 with writemask k1.
EVEX.256.0F.W1 5B /r VCVTQQ2PS xmm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert four packed quadword integers from ymm2/mem to packed single precision floating-point values in xmm1 with writemask k1.
EVEX.512.0F.W1 5B /r VCVTQQ2PS ymm1 {k1}{z}, zmm2/m512/m64bcst {er}	A	V/V	AVX512DQ OR AVX10.1	Convert eight packed quadword integers from zmm2/mem to eight packed single precision floating-point values in ymm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts packed quadword integers in the source operand (second operand) to packed single precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a YMM/XMM/XMM (lower 64 bits) register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTQQ2PS (EVEX Encoded Versions) When SRC Operand is a Register
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[k+31:k] :=
            Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[k+31:k] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[k+31:k] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

VCVTQQ2PS (EVEX Encoded Versions) When SRC Operand is a Memory Source
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[k+31:k] :=
            Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[63:0])
                ELSE
                    DEST[k+31:k] :=
            Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[k+31:k] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[k+31:k] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTQQ2PS __m256 _mm512_cvtepi64_ps(__m512i a);
VCVTQQ2PS __m256 _mm512_mask_cvtepi64_ps(__m256 s, __mmask16 k, __m512i a);
VCVTQQ2PS __m256 _mm512_maskz_cvtepi64_ps(__mmask16 k, __m512i a);
VCVTQQ2PS __m256 _mm512_cvt_roundepi64_ps(__m512i a, int r);
VCVTQQ2PS __m256 _mm512_mask_cvt_roundepi_ps(__m256 s, __mmask8 k, __m512i a, int r);
VCVTQQ2PS __m256 _mm512_maskz_cvt_roundepi64_ps(__mmask8 k, __m512i a, int r);
VCVTQQ2PS __m128 _mm256_cvtepi64_ps(__m256i a);
VCVTQQ2PS __m128 _mm256_mask_cvtepi64_ps(__m128 s, __mmask8 k, __m256i a);
VCVTQQ2PS __m128 _mm256_maskz_cvtepi64_ps(__mmask8 k, __m256i a);
VCVTQQ2PS __m128 _mm_cvtepi64_ps(__m128i a);
VCVTQQ2PS __m128 _mm_mask_cvtepi64_ps(__m128 s, __mmask8 k, __m128i a);
VCVTQQ2PS __m128 _mm_maskz_cvtepi64_ps(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”
Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTSD2SH—Convert Low FP64 Value to an FP16 Value

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F2.MAP5.W1 5A /r VCVTSD2SH xmm1{k1}{z}, xmm2, xmm3/m64 {er}	A	V/V	AVX512_FP16 OR AVX10.1	Convert the low FP64 value in xmm3/m64 to an FP16 value and store the result in the low element of xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction converts the low FP64 value in the second source operand to an FP16 value, and stores the result in the low element of the destination operand.

When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

VCVTSD2SH dest, src1, src2

```
IF *SRC2 is a register* and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    DEST.fp16[0] := Convert_fp64_to_fp16(SRC2.fp64[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else dest.fp16[0] remains unchanged

DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSD2SH __m128h _mm_cvt_roundsd_sh (__m128h a, __m128d b, const int rounding);
VCVTSD2SH __m128h _mm_mask_cvt_roundsd_sh (__m128h src, __mmask8 k, __m128h a, __m128d b, const int rounding);
VCVTSD2SH __m128h _mm_maskz_cvt_roundsd_sh (__mmask8 k, __m128h a, __m128d b, const int rounding);
VCVTSD2SH __m128h _mm_cvtsd_sh (__m128h a, __m128d b);
VCVTSD2SH __m128h _mm_mask_cvtsd_sh (__m128h src, __mmask8 k, __m128h a, __m128d b);
VCVTSD2SH __m128h _mm_maskz_cvtsd_sh (__mmask8 k, __m128h a, __m128d b);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VCVTSD2USI—Convert Scalar Double Precision Floating-Point Value to Unsigned Integer

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F2.0F.W0 79 /r VCVTSD2USI r32, xmm1/m64{er}	A	V/V	AVX512F OR AVX10.1	Convert one double precision floating-point value from xmm1/m64 to one unsigned doubleword integer r32.
EVEX.LLIG.F2.0F.W1 79 /r VCVTSD2USI r64, xmm1/m64{er}	A	V/N.E.[1]	AVX512F OR AVX10.1	Convert one double precision floating-point value from xmm1/m64 to one unsigned quadword integer zero-extended into r64.

NOTES:

1. EVEX.W1 in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts a double precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer in the destination operand (the first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFFH is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFF_FFFFFFFFH is returned.

Operation

VCVTSD2USI (EVEX Encoded Version)

```
IF (SRC *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-Bit Mode and OperandSize = 64
    THEN    DEST[63:0] := Convert_Double_Precision_Floating_Point_To_UInteger(SRC[63:0]);
    ELSE    DEST[31:0] := Convert_Double_Precision_Floating_Point_To_UInteger(SRC[63:0]);
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSD2USI unsigned int _mm_cvtsd_u32(__m128d);
VCVTSD2USI unsigned int _mm_cvt_roundsd_u32(__m128d, int r);
VCVTSD2USI unsigned __int64 _mm_cvtsd_u64(__m128d);
VCVTSD2USI unsigned __int64 _mm_cvt_roundsd_u64(__m128d, int r);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-50, “Type E3NF Class Exception Conditions.”

VCVTSH2SD—Convert Low FP16 Value to an FP64 Value

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 5A /r VCVTSH2SD xmm1{k1}{z}, xmm2, xmm3/m16 {sae}	A	V/V	AVX512_FP16 OR AVX10.1	Convert the low FP16 value in xmm3/m16 to an FP64 value and store the result in the low element of xmm1 subject to writemask k1. Bits 127:64 of xmm2 are copied to xmm1[127:64].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction converts the low FP16 element in the second source operand to a FP64 element in the low element of the destination operand.

Bits 127:64 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP64 element of the destination is updated according to the writemask.

Operation

VCVTSH2SD dest, src1, src2

```
IF k1[0] OR *no writemask*:
    DEST.fp64[0] := Convert_fp16_to_fp64(SRC2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp64[0] := 0
// else dest.fp64[0] remains unchanged

DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSH2SD __m128d _mm_cvt_roundsh_sd (__m128d a, __m128h b, const int sae);
VCVTSH2SD __m128d _mm_mask_cvt_roundsh_sd (__m128d src, __mmask8 k, __m128d a, __m128h b, const int sae);
VCVTSH2SD __m128d _mm_maskz_cvt_roundsh_sd (__mmask8 k, __m128d a, __m128h b, const int sae);
VCVTSH2SD __m128d _mm_cvtsh_sd (__m128d a, __m128h b);
VCVTSH2SD __m128d _mm_mask_cvtsh_sd (__m128d src, __mmask8 k, __m128d a, __m128h b);
VCVTSH2SD __m128d _mm_maskz_cvtsh_sd (__mmask8 k, __m128d a, __m128h b);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VCVTSH2SI—Convert Low FP16 Value to Signed Integer

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 2D /r VCVTSH2SI r32, xmm1/m16 {er}	A	V/V[1]	AVX512_FP16 OR AVX10.1	Convert the low FP16 element in xmm1/m16 to a signed doubleword integer and store the result in r32.
EVEX.LLIG.F3.MAP5.W1 2D /r VCVTSH2SI r64, xmm1/m16 {er}	A	V/N.E.	AVX512_FP16 OR AVX10.1	Convert the low FP16 element in xmm1/m16 to a signed quadword integer and store the result in r64.

NOTES:

1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts the low FP16 element in the source operand to a signed integer in the destination general purpose register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000_00000000H is returned.

Operation

VCVTSH2SI dest, src

```
IF *SRC is a register* and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

IF 64-mode and OperandSize == 64:
    DEST.qword := Convert_fp16_to_integer64(SRC.fp16[0])
ELSE:
    DEST.dword := Convert_fp16_to_integer32(SRC.fp16[0])
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSH2SI int _mm_cvt_roundsh_i32 (__m128h a, int rounding);
VCVTSH2SI __int64 _mm_cvt_roundsh_i64 (__m128h a, int rounding);
VCVTSH2SI int _mm_cvtsh_i32 (__m128h a);
VCVTSH2SI __int64 _mm_cvtsh_i64 (__m128h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-50, “Type E3NF Class Exception Conditions.”

VCVTSH2SS—Convert Low FP16 Value to FP32 Value

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.NP.MAP6.W0 13 /r VCVTSH2SS xmm1{k1}{z}, xmm2, xmm3/m16 {sae}	A	V/V	AVX512_FP16 OR AVX10.1	Convert the low FP16 element in xmm3/m16 to an FP32 value and store in the low element of xmm1 subject to writemask k1. Bits 127:32 of xmm2 are copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction converts the low FP16 element in the second source operand to the low FP32 element of the destination operand.

Bits 127:32 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

VCVTSH2SS dest, src1, src2

```
IF k1[0] OR *no writemask*:
    DEST.fp32[0] := Convert_fp16_to_fp32(SRC2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp32[0] := 0
// else dest.fp32[0] remains unchanged

DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSH2SS __m128 _mm_cvt_roundsh_ss (__m128 a, __m128h b, const int sae);
VCVTSH2SS __m128 _mm_mask_cvt_roundsh_ss (__m128 src, __mmask8 k, __m128 a, __m128h b, const int sae);
VCVTSH2SS __m128 _mm_maskz_cvt_roundsh_ss (__mmask8 k, __m128 a, __m128h b, const int sae);
VCVTSH2SS __m128 _mm_cvtsh_ss (__m128 a, __m128h b);
VCVTSH2SS __m128 _mm_mask_cvtsh_ss (__m128 src, __mmask8 k, __m128 a, __m128h b);
VCVTSH2SS __m128 _mm_maskz_cvtsh_ss (__mmask8 k, __m128 a, __m128h b);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VCVTSH2USI—Convert Low FP16 Value to Unsigned Integer

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 79 /r VCVTSH2USI r32, xmm1/m16 {er}	A	V/V[1]	AVX512_FP16 OR AVX10.1	Convert the low FP16 element in xmm1/m16 to an unsigned doubleword integer and store the result in r32.
EVEX.LLIG.F3.MAP5.W1 79 /r VCVTSH2USI r64, xmm1/m16 {er}	A	V/N.E.	AVX512_FP16 OR AVX10.1	Convert the low FP16 element in xmm1/m16 to an unsigned quadword integer and store the result in r64.

NOTES:

1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts the low FP16 element in the source operand to an unsigned integer in the destination general purpose register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFFH is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFF_FFFFFFFFH is returned.

Operation

VCVTSH2USI dest, src

```
// SET_RM() sets the rounding mode used for this instruction.
IF *SRC is a register* and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

IF 64-mode and OperandSize == 64:
    DEST.qword := Convert_fp16_to_unsigned_integer64(SRC.fp16[0])
ELSE:
    DEST.dword := Convert_fp16_to_unsigned_integer32(SRC.fp16[0])
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSH2USI unsigned int _mm_cvt_roundsh_u32 (__m128h a, int sae);
VCVTSH2USI unsigned __int64 _mm_cvt_roundsh_u64 (__m128h a, int rounding);
VCVTSH2USI unsigned int _mm_cvtsh_u32 (__m128h a);
VCVTSH2USI unsigned __int64 _mm_cvtsh_u64 (__m128h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-50, “Type E3NF Class Exception Conditions.”

VCVTSI2SH—Convert a Signed Doubleword/Quadword Integer to an FP16 Value

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 2A /r VCVTSI2SH xmm1, xmm2, r32/m32 {er}	A	V/V[1]	AVX512_FP16 OR AVX10.1	Convert the signed doubleword integer in r32/m32 to an FP16 value and store the result in xmm1. Bits 127:16 of xmm2 are copied to xmm1[127:16].
EVEX.LLIG.F3.MAP5.W1 2A /r VCVTSI2SH xmm1, xmm2, r64/m64 {er}	A	V/N.E.	AVX512_FP16 OR AVX10.1	Convert the signed quadword integer in r64/m64 to an FP16 value and store the result in xmm1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

NOTES:

1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the second source operand to an FP16 value in the destination operand. The result is stored in the low word of the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or embedded rounding controls.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers. Bits 127:16 of the XMM register destination are copied from corresponding bits in the first source operand. Bits MAXVL-1:128 of the destination register are zeroed.

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

Operation

VCVTSI2SH dest, src1, src2

```
IF *SRC2 is a register* and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

IF 64-mode and OperandSize == 64:
    DEST.fp16[0] := Convert_integer64_to_fp16(SRC2.qword)
ELSE:
    DEST.fp16[0] := Convert_integer32_to_fp16(SRC2.dword)

DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSI2SH __m128h _mm_cvt_roundi32_sh (__m128h a, int b, int rounding);
VCVTSI2SH __m128h _mm_cvt_roundi64_sh (__m128h a, __int64 b, int rounding);
VCVTSI2SH __m128h _mm_cvti32_sh (__m128h a, int b);
VCVTSI2SH __m128h _mm_cvti64_sh (__m128h a, __int64 b);

SIMD Floating-Point Exceptions

Overflow, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-50, “Type E3NF Class Exception Conditions.”

VCVTSS2SH—Convert Low FP32 Value to an FP16 Value

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.NP.MAP5.W0 1D /r VCVTSS2SH xmm1{k1}{z}, xmm2, xmm3/m32 {er}	A	V/V	AVX512_FP16 OR AVX10.1	Convert low FP32 value in xmm3/m32 to an FP16 value and store in the low element of xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction converts the low FP32 value in the second source operand to a FP16 value in the low element of the destination operand.

When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

VCVTSS2SH dest, src1, src2

```
IF *SRC2 is a register* and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    DEST.fp16[0] := Convert_fp32_to_fp16(SRC2.fp32[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else dest.fp16[0] remains unchanged

DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSS2SH __m128h _mm_cvt_roundss_sh (__m128h a, __m128 b, const int rounding);
VCVTSS2SH __m128h _mm_mask_cvt_roundss_sh (__m128h src, __mmask8 k, __m128h a, __m128 b, const int rounding);
VCVTSS2SH __m128h _mm_maskz_cvt_roundss_sh (__mmask8 k, __m128h a, __m128 b, const int rounding);
VCVTSS2SH __m128h _mm_cvtss_sh (__m128h a, __m128 b);
VCVTSS2SH __m128h _mm_mask_cvtss_sh (__m128h src, __mmask8 k, __m128h a, __m128 b);
VCVTSS2SH __m128h _mm_maskz_cvtss_sh (__mmask8 k, __m128h a, __m128 b);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VCVTSS2USI—Convert Scalar Single Precision Floating-Point Value to Unsigned Doubleword Integer

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.0F.W0 79 /r VCVTSS2USI r32, xmm1/m32{er}	A	V/V	AVX512F OR AVX10.1	Convert one single precision floating-point value from xmm1/m32 to one unsigned doubleword integer in r32.
EVEX.LLIG.F3.0F.W1 79 /r VCVTSS2USI r64, xmm1/m32{er}	A	V/N.E.[1]	AVX512F OR AVX10.1	Convert one single precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64.

NOTES:

1. EVEX.W1 in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts a single precision floating-point value in the source operand (the second operand) to an unsigned double-word integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFFH is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFF_FFFFFFFFH is returned.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTSS2USI (EVEX Encoded Version)

```
IF (SRC *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-bit Mode and OperandSize = 64
THEN
    DEST[63:0] := Convert_Single_Precision_Floating_Point_To_UInteger(SRC[31:0]);
ELSE
    DEST[31:0] := Convert_Single_Precision_Floating_Point_To_UInteger(SRC[31:0]);
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSS2USI unsigned _mm_cvtss_u32(__m128 a);
VCVTSS2USI unsigned _mm_cvt_roundss_u32(__m128 a, int r);
VCVTSS2USI unsigned __int64 _mm_cvtss_u64(__m128 a);
VCVTSS2USI unsigned __int64 _mm_cvt_roundss_u64(__m128 a, int r);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-50, "Type E3NF Class Exception Conditions."

VCVTTPD2QQ—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Quadword Integers

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W1 7A /r VCVTTPD2QQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert two packed double precision floating-point values from zmm2/m128/m64bcst to two packed signed quadword integers in zmm1 using truncation with writemask k1.
EVEX.256.66.0F.W1 7A /r VCVTTPD2QQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert four packed double precision floating-point values from ymm2/m256/m64bcst to four packed signed quadword integers in ymm1 using truncation with writemask k1.
EVEX.512.66.0F.W1 7A /r VCVTTPD2QQ zmm1 {k1}{z}, zmm2/m512/m64bcst {sae}	A	V/V	AVX512DQ OR AVX10.1	Convert eight packed double precision floating-point values from zmm2/m512 to eight packed signed quadword integers in zmm1 using truncation with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts with truncation packed double precision floating-point values in the source operand (second operand) to packed quadword integers in the destination operand (first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000_00000000H is returned.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTTPD2QQ (EVEX Encoded Version) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_Double_Precision_Floating_Point_To_QuadInteger_Truncate(SRC[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTTPD2QQ (EVEX Encoded Version) When SRC Operand is a Memory Source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] :=          Convert_Double_Precision_Floating_Point_To_QuadInteger_Truncate(SRC[63:0])
                ELSE
                    DEST[i+63:i] := Convert_Double_Precision_Floating_Point_To_QuadInteger_Truncate(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPD2QQ __m512i _mm512_cvttpd_epi64(__m512d a);
VCVTTPD2QQ __m512i _mm512_mask_cvttpd_epi64(__m512i s, __mmask8 k, __m512d a);
VCVTTPD2QQ __m512i _mm512_maskz_cvttpd_epi64(__mmask8 k, __m512d a);
VCVTTPD2QQ __m512i _mm512_cvtt_roundpd_epi64(__m512d a, int sae);
VCVTTPD2QQ __m512i _mm512_mask_cvtt_roundpd_epi64(__m512i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2QQ __m512i _mm512_maskz_cvtt_roundpd_epi64(__mmask8 k, __m512d a, int sae);
VCVTTPD2QQ __m256i _mm256_mask_cvttpd_epi64(__m256i s, __mmask8 k, __m256d a);
VCVTTPD2QQ __m256i _mm256_maskz_cvttpd_epi64(__mmask8 k, __m256d a);
VCVTTPD2QQ __m128i _mm_mask_cvttpd_epi64(__m128i s, __mmask8 k, __m128d a);
VCVTTPD2QQ __m128i _mm_maskz_cvttpd_epi64(__mmask8 k, __m128d a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTTPD2UDQ—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Unsigned Doubleword Integers

Opcode Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.0F.W1 78 /r VCVTTPD2UDQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert two packed double precision floating-point values in xmm2/m128/m64bcst to two unsigned doubleword integers in xmm1 using truncation subject to writemask k1.
EVEX.256.0F.W1 78 02 /r VCVTTPD2UDQ xmm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four unsigned doubleword integers in xmm1 using truncation subject to writemask k1.
EVEX.512.0F.W1 78 /r VCVTTPD2UDQ ymm1 {k1}{z}, zmm2/m512/m64bcst {sae}	A	V/V	AVX512F OR AVX10.1	Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight unsigned doubleword integers in ymm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts with truncation packed double precision floating-point values in the source operand (the second operand) to packed unsigned doubleword integers in the destination operand (the first operand).

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFFH is returned.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1. The upper bits (MAXVL-1:256) of the corresponding destination are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTTPD2UDQ (EVEX Encoded Versions) When SRC2 Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[k+63:k])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
```

```
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

VCVTTPD2UDQ (EVEX Encoded Versions) When SRC Operand is a Memory Source
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[k+63:k])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPD2UDQ __m256i _mm512_cvttpd_epu32(__m512d a);
VCVTTPD2UDQ __m256i _mm512_mask_cvttpd_epu32(__m256i s, __mmask8 k, __m512d a);
VCVTTPD2UDQ __m256i _mm512_maskz_cvttpd_epu32(__mmask8 k, __m512d a);
VCVTTPD2UDQ __m256i _mm512_cvtt_roundpd_epu32(__m512d a, int sae);
VCVTTPD2UDQ __m256i _mm512_mask_cvtt_roundpd_epu32(__m256i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2UDQ __m256i _mm512_maskz_cvtt_roundpd_epu32(__mmask8 k, __m512d a, int sae);
VCVTTPD2UDQ __m128i _mm256_mask_cvttpd_epu32(__m128i s, __mmask8 k, __m256d a);
VCVTTPD2UDQ __m128i _mm256_maskz_cvttpd_epu32(__mmask8 k, __m256d a);
VCVTTPD2UDQ __m128i _mm_mask_cvttpd_epu32(__m128i s, __mmask8 k, __m128d a);
VCVTTPD2UDQ __m128i _mm_maskz_cvttpd_epu32(__mmask8 k, __m128d a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."
Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTTPD2UQQ—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Unsigned Quadword Integers

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W1 78 /r VCVTTPD2UQQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert two packed double precision floating-point values from xmm2/m128/m64bcst to two packed unsigned quadword integers in xmm1 using truncation with writemask k1.
EVEX.256.66.0F.W1 78 /r VCVTTPD2UQQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert four packed double precision floating-point values from ymm2/m256/m64bcst to four packed unsigned quadword integers in ymm1 using truncation with writemask k1.
EVEX.512.66.0F.W1 78 /r VCVTTPD2UQQ zmm1 {k1}{z}, zmm2/m512/m64bcst {sae}	A	V/V	AVX512DQ OR AVX10.1	Convert eight packed double precision floating-point values from zmm2/mem to eight packed unsigned quadword integers in zmm1 using truncation with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts with truncation packed double precision floating-point values in the source operand (second operand) to packed unsigned quadword integers in the destination operand (first operand).

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFF_FFFFFFFFH is returned.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTTPD2UQQ (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_Double_Precision_Floating_Point_To_UQuadInteger_Truncate(SRC[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTTPD2UQQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] :=
            Convert_Double_Precision_Floating_Point_To_UQuadInteger_Truncate(SRC[63:0])
                ELSE
                    DEST[i+63:i] :=
            Convert_Double_Precision_Floating_Point_To_UQuadInteger_Truncate(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPD2UQQ _mm<size>[_mask[z]]_cvtt[_round]pd_epu64
VCVTTPD2UQQ __m512i _mm512_cvttpd_epu64(__m512d a);
VCVTTPD2UQQ __m512i _mm512_mask_cvttpd_epu64(__m512i s, __mmask8 k, __m512d a);
VCVTTPD2UQQ __m512i _mm512_maskz_cvttpd_epu64(__mmask8 k, __m512d a);
VCVTTPD2UQQ __m512i _mm512_cvtt_roundpd_epu64(__m512d a, int sae);
VCVTTPD2UQQ __m512i _mm512_mask_cvtt_roundpd_epu64(__m512i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2UQQ __m512i _mm512_maskz_cvtt_roundpd_epu64(__mmask8 k, __m512d a, int sae);
VCVTTPD2UQQ __m256i _mm256_mask_cvttpd_epu64(__m256i s, __mmask8 k, __m256d a);
VCVTTPD2UQQ __m256i _mm256_maskz_cvttpd_epu64(__mmask8 k, __m256d a);
VCVTTPD2UQQ __m128i _mm_mask_cvttpd_epu64(__m128i s, __mmask8 k, __m128d a);
VCVTTPD2UQQ __m128i _mm_maskz_cvttpd_epu64(__mmask8 k, __m128d a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."
Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTTPH2DQ—Convert with Truncation Packed FP16 Values to Signed Doubleword Integers

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.MAP5.W0 5B /r VCVTTPH2DQ xmm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert four packed FP16 values in xmm2/m64/m16bcst to four signed doubleword integers, and store the result in xmm1 using truncation subject to writemask k1.
EVEX.256.F3.MAP5.W0 5B /r VCVTTPH2DQ ymm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight signed doubleword integers, and store the result in ymm1 using truncation subject to writemask k1.
EVEX.512.F3.MAP5.W0 5B /r VCVTTPH2DQ zmm1{k1}{z}, ymm2/m256/m16bcst {sae}	A	V/V	AVX512_FP16 OR AVX10.1	Convert sixteen packed FP16 values in ymm2/m256/m16bcst to sixteen signed doubleword integers, and store the result in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Half	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed FP16 values in the source operand to signed doubleword integers in destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

The destination elements are updated according to the writemask.

Operation

VCVTTPH2DQ dest, src

```
VL = 128, 256 or 512
KL := VL / 32

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.fp32[j] := Convert_fp16_to_integer32_truncate(tsrc)
    ELSE IF *zeroing*:
        DEST.fp32[j] := 0
    // else dest.fp32[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPH2DQ __m512i _mm512_cvtt_roundph_epi32 (__m256h a, int sae);
VCVTTPH2DQ __m512i _mm512_mask_cvtt_roundph_epi32 (__m512i src, __mmask16 k, __m256h a, int sae);
VCVTTPH2DQ __m512i _mm512_maskz_cvtt_roundph_epi32 (__mmask16 k, __m256h a, int sae);
VCVTTPH2DQ __m128i _mm_cvttph_epi32 (__m128h a);
VCVTTPH2DQ __m128i _mm_mask_cvttph_epi32 (__m128i src, __mmask8 k, __m128h a);
VCVTTPH2DQ __m128i _mm_maskz_cvttph_epi32 (__mmask8 k, __m128h a);
VCVTTPH2DQ __m256i _mm256_cvttph_epi32 (__m128h a);
VCVTTPH2DQ __m256i _mm256_mask_cvttph_epi32 (__m256i src, __mmask8 k, __m128h a);
VCVTTPH2DQ __m256i _mm256_maskz_cvttph_epi32 (__mmask8 k, __m128h a);
VCVTTPH2DQ __m512i _mm512_cvttph_epi32 (__m256h a);
VCVTTPH2DQ __m512i _mm512_mask_cvttph_epi32 (__m512i src, __mmask16 k, __m256h a);
VCVTTPH2DQ __m512i _mm512_maskz_cvttph_epi32 (__mmask16 k, __m256h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTTPH2QQ—Convert with Truncation Packed FP16 Values to Signed Quadword Integers

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.W0 7A /r VCVTTPH2QQ xmm1{k1}{z}, xmm2/m32/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert two packed FP16 values in xmm2/m32/m16bcst to two signed quadword integers, and store the result in xmm1 using truncation subject to writemask k1.
EVEX.256.66.MAP5.W0 7A /r VCVTTPH2QQ ymm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert four packed FP16 values in xmm2/m64/m16bcst to four signed quadword integers, and store the result in ymm1 using truncation subject to writemask k1.
EVEX.512.66.MAP5.W0 7A /r VCVTTPH2QQ zmm1{k1}{z}, xmm2/m128/m16bcst {sae}	A	V/V	AVX512_FP16 OR AVX10.1	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight signed quadword integers, and store the result in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Quarter	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed FP16 values in the source operand to signed quadword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000_00000000H is returned.

The destination elements are updated according to the writemask.

Operation

VCVTTPH2QQ dest, src

```
VL = 128, 256 or 512
KL := VL / 64

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.qword[j] := Convert_fp16_to_integer64_truncate(tsrc)
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    // else dest.qword[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPH2QQ __m512i _mm512_cvtt_roundph_epi64 (__m128h a, int sae);
VCVTTPH2QQ __m512i _mm512_mask_cvtt_roundph_epi64 (__m512i src, __mmask8 k, __m128h a, int sae);
VCVTTPH2QQ __m512i _mm512_maskz_cvtt_roundph_epi64 (__mmask8 k, __m128h a, int sae);
VCVTTPH2QQ __m128i _mm_cvttph_epi64 (__m128h a);
VCVTTPH2QQ __m128i _mm_mask_cvttph_epi64 (__m128i src, __mmask8 k, __m128h a);
VCVTTPH2QQ __m128i _mm_maskz_cvttph_epi64 (__mmask8 k, __m128h a);
VCVTTPH2QQ __m256i _mm256_cvttph_epi64 (__m128h a);
VCVTTPH2QQ __m256i _mm256_mask_cvttph_epi64 (__m256i src, __mmask8 k, __m128h a);
VCVTTPH2QQ __m256i _mm256_maskz_cvttph_epi64 (__mmask8 k, __m128h a);
VCVTTPH2QQ __m512i _mm512_cvttph_epi64 (__m128h a);
VCVTTPH2QQ __m512i _mm512_mask_cvttph_epi64 (__m512i src, __mmask8 k, __m128h a);
VCVTTPH2QQ __m512i _mm512_maskz_cvttph_epi64 (__mmask8 k, __m128h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTTPH2UDQ—Convert with Truncation Packed FP16 Values to Unsigned Doubleword Integers

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 78 /r VCVTTPH2UDQ xmm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert four packed FP16 values in xmm2/m64/m16bcst to four unsigned doubleword integers, and store the result in xmm1 using truncation subject to writemask k1.
EVEX.256.NP.MAP5.W0 78 /r VCVTTPH2UDQ ymm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight unsigned doubleword integers, and store the result in ymm1 using truncation subject to writemask k1.
EVEX.512.NP.MAP5.W0 78 /r VCVTTPH2UDQ zmm1{k1}{z}, ymm2/m256/m16bcst {sae}	A	V/V	AVX512_FP16 OR AVX10.1	Convert sixteen packed FP16 values in ymm2/m256/m16bcst to sixteen unsigned doubleword integers, and store the result in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Half	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed FP16 values in the source operand to unsigned doubleword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFFH is returned.

The destination elements are updated according to the writemask.

Operation

VCVTTPH2UDQ dest, src

```
VL = 128, 256 or 512
KL := VL / 32

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.dword[j] := Convert_fp16_to_unsigned_integer32_truncate(tsrc)
    ELSE IF *zeroing*:
        DEST.dword[j] := 0
    // else dest.dword[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPH2UDQ __m512i _mm512_cvtt_roundph_epu32 (__m256h a, int sae);
VCVTTPH2UDQ __m512i _mm512_mask_cvtt_roundph_epu32 (__m512i src, __mmask16 k, __m256h a, int sae);
VCVTTPH2UDQ __m512i _mm512_maskz_cvtt_roundph_epu32 (__mmask16 k, __m256h a, int sae);
VCVTTPH2UDQ __m128i _mm_cvttph_epu32 (__m128h a);
VCVTTPH2UDQ __m128i _mm_mask_cvttph_epu32 (__m128i src, __mmask8 k, __m128h a);
VCVTTPH2UDQ __m128i _mm_maskz_cvttph_epu32 (__mmask8 k, __m128h a);
VCVTTPH2UDQ __m256i _mm256_cvttph_epu32 (__m128h a);
VCVTTPH2UDQ __m256i _mm256_mask_cvttph_epu32 (__m256i src, __mmask8 k, __m128h a);
VCVTTPH2UDQ __m256i _mm256_maskz_cvttph_epu32 (__mmask8 k, __m128h a);
VCVTTPH2UDQ __m512i _mm512_cvttph_epu32 (__m256h a);
VCVTTPH2UDQ __m512i _mm512_mask_cvttph_epu32 (__m512i src, __mmask16 k, __m256h a);
VCVTTPH2UDQ __m512i _mm512_maskz_cvttph_epu32 (__mmask16 k, __m256h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

VCVTTPH2UQQ—Convert with Truncation Packed FP16 Values to Unsigned Quadword Integers

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.W0 78 /r VCVTTPH2UQQ xmm1{k1}{z}, xmm2/m32/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert two packed FP16 values in xmm2/m32/m16bcst to two unsigned quadword integers, and store the result in xmm1 using truncation subject to writemask k1.
EVEX.256.66.MAP5.W0 78 /r VCVTTPH2UQQ ymm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert four packed FP16 values in xmm2/m64/m16bcst to four unsigned quadword integers, and store the result in ymm1 using truncation subject to writemask k1.
EVEX.512.66.MAP5.W0 78 /r VCVTTPH2UQQ zmm1{k1}{z}, xmm2/m128/m16bcst {sae}	A	V/V	AVX512_FP16 OR AVX10.1	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight unsigned quadword integers, and store the result in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Quarter	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed FP16 values in the source operand to unsigned quadword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFF_FFFFFFFFH is returned.

The destination elements are updated according to the writemask.

Operation

VCVTTPH2UQQ dest, src

```
VL = 128, 256 or 512
KL := VL / 64

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.qword[j] := Convert_fp16_to_unsigned_integer64_truncate(tsrc)
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    // else dest.qword[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPH2UQQ __m512i _mm512_cvtt_roundph_epu64 (__m128h a, int sae);
VCVTTPH2UQQ __m512i _mm512_mask_cvtt_roundph_epu64 (__m512i src, __mmask8 k, __m128h a, int sae);
VCVTTPH2UQQ __m512i _mm512_maskz_cvtt_roundph_epu64 (__mmask8 k, __m128h a, int sae);
VCVTTPH2UQQ __m128i _mm_cvttph_epu64 (__m128h a);
VCVTTPH2UQQ __m128i _mm_mask_cvttph_epu64 (__m128i src, __mmask8 k, __m128h a);
VCVTTPH2UQQ __m128i _mm_maskz_cvttph_epu64 (__mmask8 k, __m128h a);
VCVTTPH2UQQ __m256i _mm256_cvttph_epu64 (__m128h a);
VCVTTPH2UQQ __m256i _mm256_mask_cvttph_epu64 (__m256i src, __mmask8 k, __m128h a);
VCVTTPH2UQQ __m256i _mm256_maskz_cvttph_epu64 (__mmask8 k, __m128h a);
VCVTTPH2UQQ __m512i _mm512_cvttph_epu64 (__m128h a);
VCVTTPH2UQQ __m512i _mm512_mask_cvttph_epu64 (__m512i src, __mmask8 k, __m128h a);
VCVTTPH2UQQ __m512i _mm512_maskz_cvttph_epu64 (__mmask8 k, __m128h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

VCVTTPH2UW—Convert Packed FP16 Values to Unsigned Word Integers

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 7C /r VCVTTPH2UW xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight unsigned word integers, and store the result in xmm1 using truncation subject to writemask k1.
EVEX.256.NP.MAP5.W0 7C /r VCVTTPH2UW ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert sixteen packed FP16 values in ymm2/m256/m16bcst to sixteen unsigned word integers, and store the result in ymm1 using truncation subject to writemask k1.
EVEX.512.NP.MAP5.W0 7C /r VCVTTPH2UW zmm1{k1}{z}, zmm2/m512/m16bcst {sae}	A	V/V	AVX512_FP16 OR AVX10.1	Convert thirty-two packed FP16 values in zmm2/m512/m16bcst to thirty-two unsigned word integers, and store the result in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed FP16 values in the source operand to unsigned word integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFH is returned.

The destination elements are updated according to the writemask.

Operation

VCVTTPH2UW dest, src

```
VL = 128, 256 or 512
KL := VL / 16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.word[j] := Convert_fp16_to_unsigned_integer16_truncate(tsrc)
    ELSE IF *zeroing*:
        DEST.word[j] := 0
    // else dest.word[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPH2UW __m512i _mm512_cvtt_roundph_epu16 (__m512h a, int sae);
VCVTTPH2UW __m512i _mm512_mask_cvtt_roundph_epu16 (__m512i src, __mmask32 k, __m512h a, int sae);
VCVTTPH2UW __m512i _mm512_maskz_cvtt_roundph_epu16 (__mmask32 k, __m512h a, int sae);
VCVTTPH2UW __m128i _mm_cvttph_epu16 (__m128h a);
VCVTTPH2UW __m128i _mm_mask_cvttph_epu16 (__m128i src, __mmask8 k, __m128h a);
VCVTTPH2UW __m128i _mm_maskz_cvttph_epu16 (__mmask8 k, __m128h a);
VCVTTPH2UW __m256i _mm256_cvttph_epu16 (__m256h a);
VCVTTPH2UW __m256i _mm256_mask_cvttph_epu16 (__m256i src, __mmask16 k, __m256h a);
VCVTTPH2UW __m256i _mm256_maskz_cvttph_epu16 (__mmask16 k, __m256h a);
VCVTTPH2UW __m512i _mm512_cvttph_epu16 (__m512h a);
VCVTTPH2UW __m512i _mm512_mask_cvttph_epu16 (__m512i src, __mmask32 k, __m512h a);
VCVTTPH2UW __m512i _mm512_maskz_cvttph_epu16 (__mmask32 k, __m512h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTTPH2W—Convert Packed FP16 Values to Signed Word Integers

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.W0 7C /r VCVTTPH2W xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight signed word integers, and store the result in xmm1 using truncation subject to writemask k1.
EVEX.256.66.MAP5.W0 7C /r VCVTTPH2W ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert sixteen packed FP16 values in ymm2/m256/m16bcst to sixteen signed word integers, and store the result in ymm1 using truncation subject to writemask k1.
EVEX.512.66.MAP5.W0 7C /r VCVTTPH2W zmm1{k1}{z}, zmm2/m512/m16bcst {sae}	A	V/V	AVX512_FP16 OR AVX10.1	Convert thirty-two packed FP16 values in zmm2/m512/m16bcst to thirty-two signed word integers, and store the result in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed FP16 values in the source operand to signed word integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer indefinite value 8000H is returned.

The destination elements are updated according to the writemask.

Operation

VCVTTPH2W dest, src

```
VL = 128, 256 or 512
KL := VL / 16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.word[j] := Convert_fp16_to_integer16_truncate(tsrc)
    ELSE IF *zeroing*:
        DEST.word[j] := 0
    // else dest.word[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPH2W __m512i _mm512_cvtt_roundph_epi16 (__m512h a, int sae);
VCVTTPH2W __m512i _mm512_mask_cvtt_roundph_epi16 (__m512i src, __mmask32 k, __m512h a, int sae);
VCVTTPH2W __m512i _mm512_maskz_cvtt_roundph_epi16 (__mmask32 k, __m512h a, int sae);
VCVTTPH2W __m128i _mm_cvttph_epi16 (__m128h a);
VCVTTPH2W __m128i _mm_mask_cvttph_epi16 (__m128i src, __mmask8 k, __m128h a);
VCVTTPH2W __m128i _mm_maskz_cvttph_epi16 (__mmask8 k, __m128h a);
VCVTTPH2W __m256i _mm256_cvttph_epi16 (__m256h a);
VCVTTPH2W __m256i _mm256_mask_cvttph_epi16 (__m256i src, __mmask16 k, __m256h a);
VCVTTPH2W __m256i _mm256_maskz_cvttph_epi16 (__mmask16 k, __m256h a);
VCVTTPH2W __m512i _mm512_cvttph_epi16 (__m512h a);
VCVTTPH2W __m512i _mm512_mask_cvttph_epi16 (__m512i src, __mmask32 k, __m512h a);
VCVTTPH2W __m512i _mm512_maskz_cvttph_epi16 (__mmask32 k, __m512h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

VCVTTPS2QQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Signed Quadword Integer Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 7A /r VCVTTPS2QQ xmm1 {k1}{z}, xmm2/m64/m32bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed signed quadword values in xmm1 using truncation subject to writemask k1.
EVEX.256.66.0F.W0 7A /r VCVTTPS2QQ ymm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed quadword values in ymm1 using truncation subject to writemask k1.
EVEX.512.66.0F.W0 7A /r VCVTTPS2QQ zmm1 {k1}{z}, ymm2/m256/m32bcst {sae}	A	V/V	AVX512DQ OR AVX10.1	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed quadword values in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts with truncation packed single precision floating-point values in the source operand to eight signed quad-word integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000_00000000H is returned.

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64 bits) register or a 256/128/64-bit memory location. The destination operation is a vector register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTTPS2QQ (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_Single_Precision_To_QuadInteger_Truncate(SRC[k+31:k])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTTPS2QQ (EVEX Encoded Versions) When SRC Operand is a Memory Source
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] :=
            Convert_Single_Precision_To_QuadInteger_Truncate(SRC[31:0])
                ELSE
                    DEST[i+63:i] :=
            Convert_Single_Precision_To_QuadInteger_Truncate(SRC[k+31:k])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPS2QQ __m512i _mm512_cvttps_epi64(__m256 a);
VCVTTPS2QQ __m512i _mm512_mask_cvttps_epi64(__m512i s, __mmask16 k, __m256 a);
VCVTTPS2QQ __m512i _mm512_maskz_cvttps_epi64(__mmask16 k, __m256 a);
VCVTTPS2QQ __m512i _mm512_cvtt_roundps_epi64(__m256 a, int sae);
VCVTTPS2QQ __m512i _mm512_mask_cvtt_roundps_epi64(__m512i s, __mmask16 k, __m256 a, int sae);
VCVTTPS2QQ __m512i _mm512_maskz_cvtt_roundps_epi64(__mmask16 k, __m256 a, int sae);
VCVTTPS2QQ __m256i _mm256_mask_cvttps_epi64(__m256i s, __mmask8 k, __m128 a);
VCVTTPS2QQ __m256i _mm256_maskz_cvttps_epi64(__mmask8 k, __m128 a);
VCVTTPS2QQ __m128i _mm_mask_cvttps_epi64(__m128i s, __mmask8 k, __m128 a);
VCVTTPS2QQ __m128i _mm_maskz_cvttps_epi64(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."
Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTTPS2UDQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.0F.W0 78 /r VCVTTPS2UDQ xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned doubleword values in xmm1 using truncation subject to writemask k1.
EVEX.256.0F.W0 78 /r VCVTTPS2UDQ ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned doubleword values in ymm1 using truncation subject to writemask k1.
EVEX.512.0F.W0 78 /r VCVTTPS2UDQ zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	A	V/V	AVX512F OR AVX10.1	Convert sixteen packed single precision floating-point values from zmm2/m512/m32bcst to sixteen packed unsigned doubleword values in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts with truncation packed single precision floating-point values in the source operand to sixteen unsigned doubleword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFFH is returned.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTTPS2UDQ (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[i+31:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTTPS2UDQ (EVEX Encoded Versions) When SRC Operand is a Memory Source
(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[31:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPS2UDQ __m512i _mm512_cvttps_epu32(__m512 a);
VCVTTPS2UDQ __m512i _mm512_mask_cvttps_epu32(__m512i s, __mmask16 k, __m512 a);
VCVTTPS2UDQ __m512i _mm512_maskz_cvttps_epu32(__mmask16 k, __m512 a);
VCVTTPS2UDQ __m512i _mm512_cvtt_roundps_epu32(__m512 a, int sae);
VCVTTPS2UDQ __m512i _mm512_mask_cvtt_roundps_epu32(__m512i s, __mmask16 k, __m512 a, int sae);
VCVTTPS2UDQ __m512i _mm512_maskz_cvtt_roundps_epu32(__mmask16 k, __m512 a, int sae);
VCVTTPS2UDQ __m256i _mm256_mask_cvttps_epu32(__m256i s, __mmask8 k, __m256 a);
VCVTTPS2UDQ __m256i _mm256_maskz_cvttps_epu32(__mmask8 k, __m256 a);
VCVTTPS2UDQ __m128i _mm_mask_cvttps_epu32(__m128i s, __mmask8 k, __m128 a);
VCVTTPS2UDQ __m128i _mm_maskz_cvttps_epu32(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."
Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTTPS2UQQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 78 /r VCVTTPS2UQQ xmm1 {k1}{z}, xmm2/m64/m32bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed unsigned quadword values in xmm1 using truncation subject to writemask k1.
EVEX.256.66.0F.W0 78 /r VCVTTPS2UQQ ymm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned quadword values in ymm1 using truncation subject to writemask k1.
EVEX.512.66.0F.W0 78 /r VCVTTPS2UQQ zmm1 {k1}{z}, ymm2/m256/m32bcst {sae}	A	V/V	AVX512DQ OR AVX10.1	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned quadword values in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts with truncation up to eight packed single precision floating-point values in the source operand to unsigned quadword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFF_FFFFFFFFH is returned.

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64 bits) register or a 256/128/64-bit memory location. The destination operation is a vector register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTTPS2UQQ (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_Single_Precision_To_UQuadInteger_Truncate(SRC[k+31:k])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTTPS2UQQ (EVEX Encoded Versions) When SRC Operand is a Memory Source
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] :=
            Convert_Single_Precision_To_UQuadInteger_Truncate(SRC[31:0])
                ELSE
                    DEST[i+63:i] :=
            Convert_Single_Precision_To_UQuadInteger_Truncate(SRC[k+31:k])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPS2UQQ _mm<size>[_mask[z]]_cvtt[_round]ps_epu64
VCVTTPS2UQQ __m512i _mm512_cvttps_epu64(__m256 a);
VCVTTPS2UQQ __m512i _mm512_mask_cvttps_epu64(__m512i s, __mmask16 k, __m256 a);
VCVTTPS2UQQ __m512i _mm512_maskz_cvttps_epu64(__mmask16 k, __m256 a);
VCVTTPS2UQQ __m512i _mm512_cvtt_roundps_epu64(__m256 a, int sae);
VCVTTPS2UQQ __m512i _mm512_mask_cvtt_roundps_epu64(__m512i s, __mmask16 k, __m256 a, int sae);
VCVTTPS2UQQ __m512i _mm512_maskz_cvtt_roundps_epu64(__mmask16 k, __m256 a, int sae);
VCVTTPS2UQQ __m256i _mm256_mask_cvttps_epu64(__m256i s, __mmask8 k, __m128 a);
VCVTTPS2UQQ __m256i _mm256_maskz_cvttps_epu64(__mmask8 k, __m128 a);
VCVTTPS2UQQ __m128i _mm_mask_cvttps_epu64(__m128i s, __mmask8 k, __m128 a);
VCVTTPS2UQQ __m128i _mm_maskz_cvttps_epu64(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."
Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTTSD2USI—Convert With Truncation Scalar Double Precision Floating-Point Value to Unsigned Integer

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F2.0F.W0 78 /r VCVTTSD2USI r32, xmm1/m64{sae}	A	V/V	AVX512F OR AVX10.1	Convert one double precision floating-point value from xmm1/m64 to one unsigned doubleword integer r32 using truncation.
EVEX.LLIG.F2.0F.W1 78 /r VCVTTSD2USI r64, xmm1/m64{sae}	A	V/N.E.[1]	AVX512F OR AVX10.1	Convert one double precision floating-point value from xmm1/m64 to one unsigned quadword integer zero-extended into r64 using truncation.

NOTES:

1. For this specific instruction, EVEX.W in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts with truncation a double precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, a truncated (round toward zero) value is returned.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFFH is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFF_FFFFFFFFH is returned.

EVEX.W1 version: promotes the instruction to produce 64-bit data in 64-bit mode.

Operation

VCVTTSD2USI (EVEX Encoded Version)

```
IF 64-Bit Mode and OperandSize = 64
    THEN    DEST[63:0] := Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0]);
    ELSE    DEST[31:0] := Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0]);
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTSD2USI unsigned int _mm_cvttsd_u32(__m128d);
VCVTTSD2USI unsigned int _mm_cvtt_roundsd_u32(__m128d, int sae);
VCVTTSD2USI unsigned __int64 _mm_cvttsd_u64(__m128d);
VCVTTSD2USI unsigned __int64 _mm_cvtt_roundsd_u64(__m128d, int sae);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-50, “Type E3NF Class Exception Conditions.”

VCVTTSH2SI—Convert with Truncation Low FP16 Value to a Signed Integer

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 2C /r VCVTTSH2SI r32, xmm1/m16 {sae}	A	V/V[1]	AVX512_FP16 OR AVX10.1	Convert FP16 value in the low element of xmm1/m16 to a signed doubleword integer and store the result in r32 using truncation.
EVEX.LLIG.F3.MAP5.W1 2C /r VCVTTSH2SI r64, xmm1/m16 {sae}	A	V/N.E.	AVX512_FP16 OR AVX10.1	Convert FP16 value in the low element of xmm1/m16 to a signed quadword integer and store the result in r64 using truncation.

NOTES:

1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts the low FP16 element in the source operand to a signed integer in the destination general purpose register.

When a conversion is inexact, a truncated (round toward zero) value is returned.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000_00000000H is returned.

Operation

VCVTTSH2SI dest, src

```
IF 64-mode and OperandSize == 64:
    DEST.qword := Convert_fp16_to_integer64_truncate(SRC.fp16[0])
ELSE:
    DEST.dword := Convert_fp16_to_integer32_truncate(SRC.fp16[0])
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTSH2SI int _mm_cvtt_roundsh_i32 (__m128h a, int sae);
VCVTTSH2SI __int64 _mm_cvtt_roundsh_i64 (__m128h a, int sae);
VCVTTSH2SI int _mm_cvttsh_i32 (__m128h a);
VCVTTSH2SI __int64 _mm_cvttsh_i64 (__m128h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-50, “Type E3NF Class Exception Conditions.”

VCVTTSH2USI—Convert with Truncation Low FP16 Value to an Unsigned Integer

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 78 /r VCVTTSH2USI r32, xmm1/m16 {sae}	A	V/V[1]	AVX512_FP16 OR AVX10.1	Convert FP16 value in the low element of xmm1/m16 to an unsigned doubleword integer and store the result in r32 using truncation.
EVEX.LLIG.F3.MAP5.W1 78 /r VCVTTSH2USI r64, xmm1/m16 {sae}	A	V/N.E.	AVX512_FP16 OR AVX10.1	Convert FP16 value in the low element of xmm1/m16 to an unsigned quadword integer and store the result in r64 using truncation.

NOTES:

1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts the low FP16 element in the source operand to an unsigned integer in the destination general purpose register.

When a conversion is inexact, a truncated (round toward zero) value is returned.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFFH is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFF_FFFFFFFFH is returned.

Operation

VCVTTSH2USI dest, src

```
IF 64-mode and OperandSize == 64:
    DEST.qword := Convert_fp16_to_unsigned_integer64_truncate(SRC.fp16[0])
ELSE:
    DEST.dword := Convert_fp16_to_unsigned_integer32_truncate(SRC.fp16[0])
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTSH2USI unsigned int _mm_cvtt_roundsh_u32 (__m128h a, int sae);
VCVTTSH2USI unsigned __int64 _mm_cvtt_roundsh_u64 (__m128h a, int sae);
VCVTTSH2USI unsigned int _mm_cvttsh_u32 (__m128h a);
VCVTTSH2USI unsigned __int64 _mm_cvttsh_u64 (__m128h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-50, “Type E3NF Class Exception Conditions.”

VCVTTSS2USI—Convert With Truncation Scalar Single Precision Floating-Point Value to Unsigned Integer

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.0F.W0 78 /r VCVTTSS2USI r32, xmm1/m32{sae}	A	V/V	AVX512F OR AVX10.1	Convert one single precision floating-point value from xmm1/m32 to one unsigned doubleword integer in r32 using truncation.
EVEX.LLIG.F3.0F.W1 78 /r VCVTTSS2USI r64, xmm1/m32{sae}	A	V/N.E.[1]	AVX512F OR AVX10.1	Convert one single precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64 using truncation.

NOTES:

1. For this specific instruction, EVEX.W in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts with truncation a single precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, a truncated (round toward zero) value is returned.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFFH is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFF_FFFFFFFFH is returned.

EVEX.W1 version: promotes the instruction to produce 64-bit data in 64-bit mode.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTTSS2USI (EVEX Encoded Version)

```
IF 64-bit Mode and OperandSize = 64
THEN
    DEST[63:0] := Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[31:0]);
ELSE
    DEST[31:0] := Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[31:0]);
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTSS2USI unsigned int _mm_cvttss_u32(__m128 a);
VCVTTSS2USI unsigned int _mm_cvtt_roundss_u32(__m128 a, int sae);
VCVTTSS2USI unsigned __int64 _mm_cvttss_u64(__m128 a);
VCVTTSS2USI unsigned __int64 _mm_cvtt_roundss_u64(__m128 a, int sae);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-50, "Type E3NF Class Exception Conditions."

VCVTUDQ2PD—Convert Packed Unsigned Doubleword Integers to Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F.W0 7A /r VCVTUDQ2PD xmm1 {k1}{z}, xmm2/m64/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert two packed unsigned doubleword integers from xmm2/m64/m32bcst to packed double precision floating-point values in xmm1 with writemask k1.
EVEX.256.F3.0F.W0 7A /r VCVTUDQ2PD ymm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed unsigned doubleword integers from xmm2/m128/m32bcst to packed double precision floating-point values in ymm1 with writemask k1.
EVEX.512.F3.0F.W0 7A /r VCVTUDQ2PD zmm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512F OR AVX10.1	Convert eight packed unsigned doubleword integers from ymm2/m256/m32bcst to eight packed double precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts packed unsigned doubleword integers in the source operand (second operand) to packed double precision floating-point values in the destination operand (first operand).

The source operand is a YMM/XMM/XMM (low 64 bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Attempt to encode this instruction with EVEX embedded rounding is ignored.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTUDQ2PD (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_UInteger_To_Double_Precision_Floating_Point(SRC[k+31:k])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTUDQ2PD (EVEX Encoded Versions) When SRC Operand is a Memory Source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            Convert_UInteger_To_Double_Precision_Floating_Point(SRC[31:0])
                ELSE
                    DEST[i+63:i] :=
            Convert_UInteger_To_Double_Precision_Floating_Point(SRC[k+31:k])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTUDQ2PD __m512d _mm512_cvtepu32_pd( __m256i a);
VCVTUDQ2PD __m512d _mm512_mask_cvtepu32_pd( __m512d s, __mmask8 k, __m256i a);
VCVTUDQ2PD __m512d _mm512_maskz_cvtepu32_pd( __mmask8 k, __m256i a);
VCVTUDQ2PD __m256d _mm256_cvtepu32_pd( __m128i a);
VCVTUDQ2PD __m256d _mm256_mask_cvtepu32_pd( __m256d s, __mmask8 k, __m128i a);
VCVTUDQ2PD __m256d _mm256_maskz_cvtepu32_pd( __mmask8 k, __m128i a);
VCVTUDQ2PD __m128d _mm_cvtepu32_pd( __m128i a);
VCVTUDQ2PD __m128d _mm_mask_cvtepu32_pd( __m128d s, __mmask8 k, __m128i a);
VCVTUDQ2PD __m128d _mm_maskz_cvtepu32_pd( __mmask8 k, __m128i a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instructions, see Table 2-53, "Type E5 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTUDQ2PH—Convert Packed Unsigned Doubleword Integers to Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.MAP5.W0 7A /r VCVTUDQ2PH xmm1{k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert four packed unsigned doubleword integers from xmm2/m128/m32bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.256.F2.MAP5.W0 7A /r VCVTUDQ2PH xmm1{k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert eight packed unsigned doubleword integers from ymm2/m256/m32bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.512.F2.MAP5.W0 7A /r VCVTUDQ2PH ymm1{k1}{z}, zmm2/m512/m32bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Convert sixteen packed unsigned doubleword integers from zmm2/m512/m32bcst to packed FP16 values, and store the result in ymm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed unsigned doubleword integers in the source operand to packed FP16 values in the destination operand. The destination elements are updated according to the writemask.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

Operation

VCVTUDQ2PH dest, src

```
VL = 128, 256 or 512
KL := VL / 32

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.dword[0]
        ELSE
            tsrc := SRC.dword[j]
        DEST.fp16[j] := Convert_unsigned_integer32_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL/2] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUDQ2PH __m256h _mm512_cvt_roundepu32_ph (__m512i a, int rounding);
VCVTUDQ2PH __m256h _mm512_mask_cvt_roundepu32_ph (__m256h src, __mmask16 k, __m512i a, int rounding);
VCVTUDQ2PH __m256h _mm512_maskz_cvt_roundepu32_ph (__mmask16 k, __m512i a, int rounding);
VCVTUDQ2PH __m128h _mm_cvtepu32_ph (__m128i a);
VCVTUDQ2PH __m128h _mm_mask_cvtepu32_ph (__m128h src, __mmask8 k, __m128i a);
VCVTUDQ2PH __m128h _mm_maskz_cvtepu32_ph (__mmask8 k, __m128i a);
VCVTUDQ2PH __m128h _mm256_cvtepu32_ph (__m256i a);
VCVTUDQ2PH __m128h _mm256_mask_cvtepu32_ph (__m128h src, __mmask8 k, __m256i a);
VCVTUDQ2PH __m128h _mm256_maskz_cvtepu32_ph (__mmask8 k, __m256i a);
VCVTUDQ2PH __m256h _mm512_cvtepu32_ph (__m512i a);
VCVTUDQ2PH __m256h _mm512_mask_cvtepu32_ph (__m256h src, __mmask16 k, __m512i a);
VCVTUDQ2PH __m256h _mm512_maskz_cvtepu32_ph (__mmask16 k, __m512i a);

SIMD Floating-Point Exceptions

Overflow, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

VCVTUDQ2PS—Convert Packed Unsigned Doubleword Integers to Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.0F.W0 7A /r VCVTUDQ2PS xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed unsigned doubleword integers from xmm2/m128/m32bcst to packed single precision floating-point values in xmm1 with writemask k1.
EVEX.256.F2.0F.W0 7A /r VCVTUDQ2PS ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert eight packed unsigned doubleword integers from ymm2/m256/m32bcst to packed single precision floating-point values in ymm1 with writemask k1.
EVEX.512.F2.0F.W0 7A /r VCVTUDQ2PS zmm1 {k1}{z}, zmm2/m512/m32bcst {er}	A	V/V	AVX512F OR AVX10.1	Convert sixteen packed unsigned doubleword integers from zmm2/m512/m32bcst to sixteen packed single precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts packed unsigned doubleword integers in the source operand (second operand) to single precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTUDQ2PS (EVEX Encoded Version) When SRC Operand is a Register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            Convert_UInteger_To_Single_Precision_Floating_Point(SRC[i+31:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
```

```
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTUDQ2PS (EVEX Encoded Version) When SRC Operand is a Memory Source
(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            Convert_UInteger_To_Single_Precision_Floating_Point(SRC[31:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_UInteger_To_Single_Precision_Floating_Point(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUDQ2PS __m512 _mm512_cvtepu32_ps(__m512i a);
VCVTUDQ2PS __m512 _mm512_mask_cvtepu32_ps(__m512 s, __mmask16 k, __m512i a);
VCVTUDQ2PS __m512 _mm512_maskz_cvtepu32_ps(__mmask16 k, __m512i a);
VCVTUDQ2PS __m512 _mm512_cvt_roundepu32_ps(__m512i a, int r);
VCVTUDQ2PS __m512 _mm512_mask_cvt_roundepu32_ps(__m512 s, __mmask16 k, __m512i a, int r);
VCVTUDQ2PS __m512 _mm512_maskz_cvt_roundepu32_ps(__mmask16 k, __m512i a, int r);
VCVTUDQ2PS __m256 _mm256_cvtepu32_ps(__m256i a);
VCVTUDQ2PS __m256 _mm256_mask_cvtepu32_ps(__m256 s, __mmask8 k, __m256i a);
VCVTUDQ2PS __m256 _mm256_maskz_cvtepu32_ps(__mmask8 k, __m256i a);
VCVTUDQ2PS __m128 _mm_cvtepu32_ps(__m128i a);
VCVTUDQ2PS __m128 _mm_mask_cvtepu32_ps(__m128 s, __mmask8 k, __m128i a);
VCVTUDQ2PS __m128 _mm_maskz_cvtepu32_ps(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”
Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTUQQ2PD—Convert Packed Unsigned Quadword Integers to Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F.W1 7A /r VCVTUQQ2PD xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert two packed unsigned quadword integers from xmm2/m128/m64bcst to two packed double precision floating-point values in xmm1 with writemask k1.
EVEX.256.F3.0F.W1 7A /r VCVTUQQ2PD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert four packed unsigned quadword integers from ymm2/m256/m64bcst to packed double precision floating-point values in ymm1 with writemask k1.
EVEX.512.F3.0F.W1 7A /r VCVTUQQ2PD zmm1 {k1}{z}, zmm2/m512/m64bcst {er}	A	V/V	AVX512DQ OR AVX10.1	Convert eight packed unsigned quadword integers from zmm2/m512/m64bcst to eight packed double precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts packed unsigned quadword integers in the source operand (second operand) to packed double precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTUQQ2PD (EVEX Encoded Version) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL == 512) AND (EVEX.b == 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
```

```
DEST[MAXVL-1:VL] := 0
```

VCVTUQQ2PD (EVEX Encoded Version) When SRC Operand is a Memory Source
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] :=
            Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[63:0])
                ELSE
                    DEST[i+63:i] :=
            Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTUQQ2PD __m512d _mm512_cvtepu64_ps( __m512i a);
VCVTUQQ2PD __m512d _mm512_mask_cvtepu64_ps( __m512d s, __mmask8 k, __m512i a);
VCVTUQQ2PD __m512d _mm512_maskz_cvtepu64_ps( __mmask8 k, __m512i a);
VCVTUQQ2PD __m512d _mm512_cvt_roundepu64_ps( __m512i a, int r);
VCVTUQQ2PD __m512d _mm512_mask_cvt_roundepu64_ps( __m512d s, __mmask8 k, __m512i a, int r);
VCVTUQQ2PD __m512d _mm512_maskz_cvt_roundepu64_ps( __mmask8 k, __m512i a, int r);
VCVTUQQ2PD __m256d _mm256_cvtepu64_ps( __m256i a);
VCVTUQQ2PD __m256d _mm256_mask_cvtepu64_ps( __m256d s, __mmask8 k, __m256i a);
VCVTUQQ2PD __m256d _mm256_maskz_cvtepu64_ps( __mmask8 k, __m256i a);
VCVTUQQ2PD __m128d _mm_cvtepu64_ps( __m128i a);
VCVTUQQ2PD __m128d _mm_mask_cvtepu64_ps( __m128d s, __mmask8 k, __m128i a);
VCVTUQQ2PD __m128d _mm_maskz_cvtepu64_ps( __mmask8 k, __m128i a);
```

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."
Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTUQQ2PH—Convert Packed Unsigned Quadword Integers to Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.MAP5.W1 7A /r VCVTUQQ2PH xmm1{k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert two packed unsigned quadword integers from xmm2/m128/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.256.F2.MAP5.W1 7A /r VCVTUQQ2PH xmm1{k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert four packed unsigned quadword integers from ymm2/m256/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.512.F2.MAP5.W1 7A /r VCVTUQQ2PH xmm1{k1}{z}, zmm2/m512/m64bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Convert eight packed unsigned quadword integers from zmm2/m512/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed unsigned quadword integers in the source operand to packed FP16 values in the destination operand. The destination elements are updated according to the writemask.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

Operation

VCVTUQQ2PH dest, src

```
VL = 128, 256 or 512
KL := VL / 64

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.qword[0]
        ELSE
            tsrc := SRC.qword[j]
        DEST.fp16[j] := Convert_unsigned_integer64_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL/4] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUQQ2PH __m128h _mm512_cvt_roundepu64_ph (__m512i a, int rounding);
VCVTUQQ2PH __m128h _mm512_mask_cvt_roundepu64_ph (__m128h src, __mmask8 k, __m512i a, int rounding);
VCVTUQQ2PH __m128h _mm512_maskz_cvt_roundepu64_ph (__mmask8 k, __m512i a, int rounding);
VCVTUQQ2PH __m128h _mm_cvtepu64_ph (__m128i a);
VCVTUQQ2PH __m128h _mm_mask_cvtepu64_ph (__m128h src, __mmask8 k, __m128i a);
VCVTUQQ2PH __m128h _mm_maskz_cvtepu64_ph (__mmask8 k, __m128i a);
VCVTUQQ2PH __m128h _mm256_cvtepu64_ph (__m256i a);
VCVTUQQ2PH __m128h _mm256_mask_cvtepu64_ph (__m128h src, __mmask8 k, __m256i a);
VCVTUQQ2PH __m128h _mm256_maskz_cvtepu64_ph (__mmask8 k, __m256i a);
VCVTUQQ2PH __m128h _mm512_cvtepu64_ph (__m512i a);
VCVTUQQ2PH __m128h _mm512_mask_cvtepu64_ph (__m128h src, __mmask8 k, __m512i a);
VCVTUQQ2PH __m128h _mm512_maskz_cvtepu64_ph (__mmask8 k, __m512i a);

SIMD Floating-Point Exceptions

Overflow, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTUQQ2PS—Convert Packed Unsigned Quadword Integers to Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.0F.W1 7A /r VCVTUQQ2PS xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert two packed unsigned quadword integers from xmm2/m128/m64bcst to packed single precision floating-point values in xmm1 with writemask k1.
EVEX.256.F2.0F.W1 7A /r VCVTUQQ2PS xmm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert four packed unsigned quadword integers from ymm2/m256/m64bcst to packed single precision floating-point values in xmm1 with writemask k1.
EVEX.512.F2.0F.W1 7A /r VCVTUQQ2PS ymm1 {k1}{z}, zmm2/m512/m64bcst {er}	A	V/V	AVX512DQ OR AVX10.1	Convert eight packed unsigned quadword integers from zmm2/m512/m64bcst to eight packed single precision floating-point values in ymm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts packed unsigned quadword integers in the source operand (second operand) to single precision floating-point values in the destination operand (first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTUQQ2PS (EVEX Encoded Version) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            Convert_UQuadInteger_To_Single_Precision_Floating_Point(SRC[k+63:k])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

VCVTUQQ2PS (EVEX Encoded Version) When SRC Operand is a Memory Source
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            Convert_UQuadInteger_To_Single_Precision_Floating_Point(SRC[63:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_UQuadInteger_To_Single_Precision_Floating_Point(SRC[k+63:k])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUQQ2PS __m256 _mm512_cvtepu64_ps(__m512i a);
VCVTUQQ2PS __m256 _mm512_mask_cvtepu64_ps(__m256 s, __mmask8 k, __m512i a);
VCVTUQQ2PS __m256 _mm512_maskz_cvtepu64_ps(__mmask8 k, __m512i a);
VCVTUQQ2PS __m256 _mm512_cvt_roundepu64_ps(__m512i a, int r);
VCVTUQQ2PS __m256 _mm512_mask_cvt_roundepu64_ps(__m256 s, __mmask8 k, __m512i a, int r);
VCVTUQQ2PS __m256 _mm512_maskz_cvt_roundepu64_ps(__mmask8 k, __m512i a, int r);
VCVTUQQ2PS __m128 _mm256_cvtepu64_ps(__m256i a);
VCVTUQQ2PS __m128 _mm256_mask_cvtepu64_ps(__m128 s, __mmask8 k, __m256i a);
VCVTUQQ2PS __m128 _mm256_maskz_cvtepu64_ps(__mmask8 k, __m256i a);
VCVTUQQ2PS __m128 _mm_cvtepu64_ps(__m128i a);
VCVTUQQ2PS __m128 _mm_mask_cvtepu64_ps(__m128 s, __mmask8 k, __m128i a);
VCVTUQQ2PS __m128 _mm_maskz_cvtepu64_ps(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."
Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTUSI2SD—Convert Unsigned Integer to Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F2.0F.W0 7B /r VCVTUSI2SD xmm1, xmm2, r/m32	A	V/V	AVX512F OR AVX10.1	Convert one unsigned doubleword integer from r/m32 to one double precision floating-point value in xmm1.
EVEX.LLIG.F2.0F.W1 7B /r VCVTUSI2SD xmm1, xmm2, r/m64{er}	A	V/N.E.[1]	AVX512F OR AVX10.1	Convert one unsigned quadword integer from r/m64 to one double precision floating-point value in xmm1.

NOTES:

1. For this specific instruction, EVEX.W in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Converts an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the second source operand to a double precision floating-point value in the destination operand. The result is stored in the low quadword of the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.W1 version: promotes the instruction to use 64-bit input value in 64-bit mode.

EVEX.W0 version: attempt to encode this instruction with EVEX embedded rounding is ignored.

Operation

VCVTUSI2SD (EVEX Encoded Version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[63:0] := Convert_UInteger_To_Double_Precision_Floating_Point(SRC2[63:0]);
ELSE
    DEST[63:0] := Convert_UInteger_To_Double_Precision_Floating_Point(SRC2[31:0]);
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUSI2SD __m128d _mm_cvtu32_sd(__m128d s, unsigned a);
VCVTUSI2SD __m128d _mm_cvtu64_sd(__m128d s, unsigned __int64 a);
VCVTUSI2SD __m128d _mm_cvt_roundu64_sd(__m128d s, unsigned __int64 a, int r);

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

See Table 2-50, "Type E3NF Class Exception Conditions" if W1; otherwise, see Table 2-61, "Type E10NF Class Exception Conditions."

VCVTUSI2SH—Convert Unsigned Doubleword Integer to an FP16 Value

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 7B /r VCVTUSI2SH xmm1, xmm2, r32/m32 {er}	A	V/V[1]	AVX512_FP16	Convert an unsigned doubleword integer from r32/m32 to an FP16 value, and store the result in xmm1. Bits 127:16 from xmm2 are copied to xmm1[127:16].
EVEX.LLIG.F3.MAP5.W1 7B /r VCVTUSI2SH xmm1, xmm2, r64/m64 {er}	A	V/N.E.	AVX512_FP16	Convert an unsigned quadword integer from r64/m64 to an FP16 value, and store the result in xmm1. Bits 127:16 from xmm2 are copied to xmm1[127:16].

NOTES:

1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction converts an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the second source operand to a FP16 value in the destination operand. The result is stored in the low word of the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or embedded rounding controls.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers. Bits 127:16 of the XMM register destination are copied from corresponding bits in the first source operand. Bits MAXVL-1:128 of the destination register are zeroed.

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

Operation

VCVTUSI2SH dest, src1, src2

```
IF *SRC2 is a register* and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

IF 64-mode and OperandSize == 64:
    DEST.fp16[0] := Convert_unsigned_integer64_to_fp16(SRC2.qword)
ELSE:
    DEST.fp16[0] := Convert_unsigned_integer32_to_fp16(SRC2.dword)

DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUSI2SH __m128h _mm_cvt_roundu32_sh (__m128h a, unsigned int b, int rounding);
VCVTUSI2SH __m128h _mm_cvt_roundu64_sh (__m128h a, unsigned __int64 b, int rounding);
VCVTUSI2SH __m128h _mm_cvtu32_sh (__m128h a, unsigned int b);
VCVTUSI2SH __m128h _mm_cvtu64_sh (__m128h a, unsigned __int64 b);

SIMD Floating-Point Exceptions

Overflow, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-50, “Type E3NF Class Exception Conditions.”

VCVTUSI2SS—Convert Unsigned Integer to Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.0F.W0 7B /r VCVTUSI2SS xmm1, xmm2, r/m32{er}	A	V/V	AVX512F OR AVX10.1	Convert one unsigned doubleword integer from r/m32 to one single precision floating-point value in xmm1.
EVEX.LLIG.F3.0F.W1 7B /r VCVTUSI2SS xmm1, xmm2, r/m64{er}	A	V/N.E.[1]	AVX512F OR AVX10.1	Convert one unsigned quadword integer from r/m64 to one single precision floating-point value in xmm1.

NOTES:

1. For this specific instruction, EVEX.W in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Converts an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the source operand (second operand) to a single precision floating-point value in the destination operand (first operand). The source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.W1 version: promotes the instruction to use 64-bit input value in 64-bit mode.

Operation

VCVTUSI2SS (EVEX Encoded Version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[31:0] := Convert_UInteger_To_Single_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[31:0] := Convert_UInteger_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUSI2SS __m128 _mm_cvtu32_ss(__m128 s, unsigned a);
VCVTUSI2SS __m128 _mm_cvt_roundu32_ss(__m128 s, unsigned a, int r);
VCVTUSI2SS __m128 _mm_cvtu64_ss(__m128 s, unsigned __int64 a);
VCVTUSI2SS __m128 _mm_cvt_roundu64_ss(__m128 s, unsigned __int64 a, int r);

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

See Table 2-50, "Type E3NF Class Exception Conditions."

VCVTUW2PH—Convert Packed Unsigned Word Integers to FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.MAP5.W0 7D /r VCVTUW2PH xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert eight packed unsigned word integers from xmm2/m128/m16bcst to FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.256.F2.MAP5.W0 7D /r VCVTUW2PH ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert sixteen packed unsigned word integers from ymm2/m256/m16bcst to FP16 values, and store the result in ymm1 subject to writemask k1.
EVEX.512.F2.MAP5.W0 7D /r VCVTUW2PH zmm1{k1}{z}, zmm2/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Convert thirty-two packed unsigned word integers from zmm2/m512/m16bcst to FP16 values, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed unsigned word integers in the source operand to FP16 values in the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or embedded rounding controls.

The destination elements are updated according to the writemask.

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

Operation

VCVTUW2PH dest, src

```
VL = 128, 256 or 512
KL := VL / 16

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.word[0]
        ELSE
            tsrc := SRC.word[j]
        DEST.fp16[j] := Convert_unsignd_integer16_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUW2PH __m512h _mm512_cvt_roundepu16_ph (__m512i a, int rounding);
VCVTUW2PH __m512h _mm512_mask_cvt_roundepu16_ph (__m512h src, __mmask32 k, __m512i a, int rounding);
VCVTUW2PH __m512h _mm512_maskz_cvt_roundepu16_ph (__mmask32 k, __m512i a, int rounding);
VCVTUW2PH __m128h _mm_cvtepu16_ph (__m128i a);
VCVTUW2PH __m128h _mm_mask_cvtepu16_ph (__m128h src, __mmask8 k, __m128i a);
VCVTUW2PH __m128h _mm_maskz_cvtepu16_ph (__mmask8 k, __m128i a);
VCVTUW2PH __m256h _mm256_cvtepu16_ph (__m256i a);
VCVTUW2PH __m256h _mm256_mask_cvtepu16_ph (__m256h src, __mmask16 k, __m256i a);
VCVTUW2PH __m256h _mm256_maskz_cvtepu16_ph (__mmask16 k, __m256i a);
VCVTUW2PH __m512h _mm512_cvtepu16_ph (__m512i a);
VCVTUW2PH __m512h _mm512_mask_cvtepu16_ph (__m512h src, __mmask32 k, __m512i a);
VCVTUW2PH __m512h _mm512_maskz_cvtepu16_ph (__mmask32 k, __m512i a);

SIMD Floating-Point Exceptions

Overflow, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

VCVTW2PH—Convert Packed Signed Word Integers to FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.MAP5.W0 7D /r VCVTW2PH xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert eight packed signed word integers from xmm2/m128/m16bcst to FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.256.F3.MAP5.W0 7D /r VCVTW2PH ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert sixteen packed signed word integers from ymm2/m256/m16bcst to FP16 values, and store the result in ymm1 subject to writemask k1.
EVEX.512.F3.MAP5.W0 7D /r VCVTW2PH zmm1{k1}{z}, zmm2/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Convert thirty-two packed signed word integers from zmm2/m512/m16bcst to FP16 values, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed signed word integers in the source operand to FP16 values in the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or embedded rounding controls.

The destination elements are updated according to the writemask.

Operation

VCVTW2PH dest, src

```
VL = 128, 256 or 512
KL := VL / 16

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.word[0]
        ELSE
            tsrc := SRC.word[j]
        DEST.fp16[j] := Convert_integer16_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTW2PH __m512h _mm512_cvt_roundepi16_ph (__m512i a, int rounding);
VCVTW2PH __m512h _mm512_mask_cvt_roundepi16_ph (__m512h src, __mmask32 k, __m512i a, int rounding);
VCVTW2PH __m512h _mm512_maskz_cvt_roundepi16_ph (__mmask32 k, __m512i a, int rounding);
VCVTW2PH __m128h _mm_cvtepi16_ph (__m128i a);
VCVTW2PH __m128h _mm_mask_cvtepi16_ph (__m128h src, __mmask8 k, __m128i a);
VCVTW2PH __m128h _mm_maskz_cvtepi16_ph (__mmask8 k, __m128i a);
VCVTW2PH __m256h _mm256_cvtepi16_ph (__m256i a);
VCVTW2PH __m256h _mm256_mask_cvtepi16_ph (__m256h src, __mmask16 k, __m256i a);
VCVTW2PH __m256h _mm256_maskz_cvtepi16_ph (__mmask16 k, __m256i a);
VCVTW2PH __m512h _mm512_cvtepi16_ph (__m512i a);
VCVTW2PH __m512h _mm512_mask_cvtepi16_ph (__m512h src, __mmask32 k, __m512i a);
VCVTW2PH __m512h _mm512_maskz_cvtepi16_ph (__mmask32 k, __m512i a);

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VDBPSADBW—Double Block Packed Sum-Absolute-Differences (SAD) on Unsigned Bytes

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 42 /r ib VDBPSADBW xmm1 {k1}{z}, xmm2, xmm3/m128, imm8	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compute packed SAD word results of unsigned bytes in dword block from xmm2 with unsigned bytes of dword blocks transformed from xmm3/m128 using the shuffle controls in imm8. Results are written to xmm1 under the writemask k1.
EVEX.256.66.0F3A.W0 42 /r ib VDBPSADBW ymm1 {k1}{z}, ymm2, ymm3/m256, imm8	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compute packed SAD word results of unsigned bytes in dword block from ymm2 with unsigned bytes of dword blocks transformed from ymm3/m256 using the shuffle controls in imm8. Results are written to ymm1 under the writemask k1.
EVEX.512.66.0F3A.W0 42 /r ib VDBPSADBW zmm1 {k1}{z}, zmm2, zmm3/m512, imm8	A	V/V	AVX512BW OR AVX10.1	Compute packed SAD word results of unsigned bytes in dword block from zmm2 with unsigned bytes of dword blocks transformed from zmm3/m512 using the shuffle controls in imm8. Results are written to zmm1 under the writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Compute packed SAD (sum of absolute differences) word results of unsigned bytes from two 32-bit dword elements. Packed SAD word results are calculated in multiples of qword superblocks, producing 4 SAD word results in each 64-bit superblock of the destination register.

Within each super block of packed word results, the SAD results from two 32-bit dword elements are calculated as follows:

- The lower two word results are calculated each from the SAD operation between a sliding dword element within a qword superblock from an intermediate vector with a stationary dword element in the corresponding qword superblock of the first source operand. The intermediate vector, see "Tmp1" in Figure 5-8, is constructed from the second source operand the imm8 byte as shuffle control to select dword elements within a 128-bit lane of the second source operand. The two sliding dword elements in a qword superblock of Tmp1 are located at byte offset 0 and 1 within the superblock, respectively. The stationary dword element in the qword superblock from the first source operand is located at byte offset 0.
- The next two word results are calculated each from the SAD operation between a sliding dword element within a qword superblock from the intermediate vector Tmp1 with a second stationary dword element in the corresponding qword superblock of the first source operand. The two sliding dword elements in a qword superblock of Tmp1 are located at byte offset 2and 3 within the superblock, respectively. The stationary dword element in the qword superblock from the first source operand is located at byte offset 4.
- The intermediate vector is constructed in 128-bits lanes. Within each 128-bit lane, each dword element of the intermediate vector is selected by a two-bit field within the imm8 byte on the corresponding 128-bits of the second source operand. The imm8 byte serves as dword shuffle control within each 128-bit lanes of the intermediate vector and the second source operand, similarly to PSHUFD.

The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1 at 16-bit word granularity.

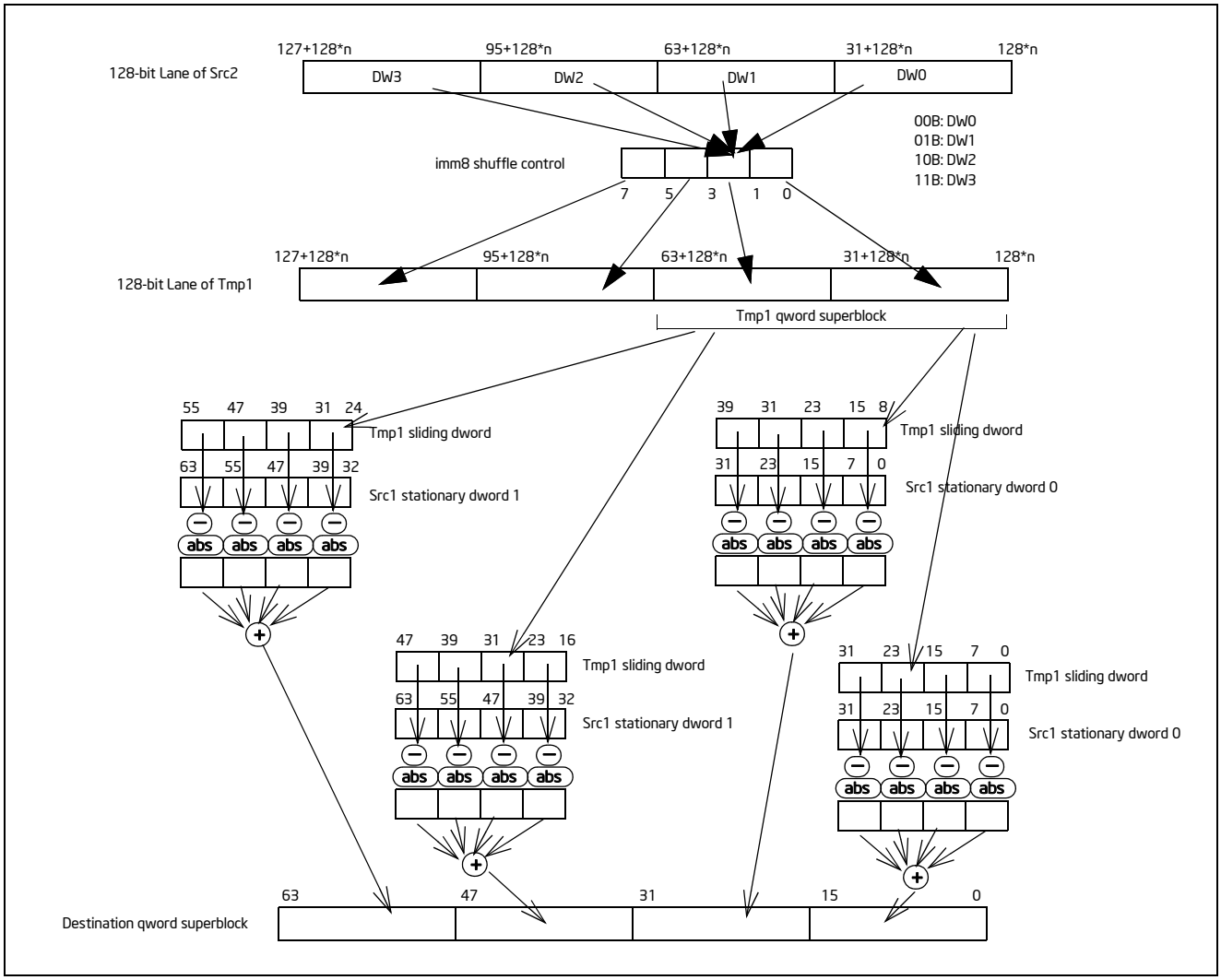


Figure 5-8. 64-bit Super Block of SAD Operation in VDBPSADBW

Operation

VDBPSADBW (EVEX Encoded Versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)
Selection of quadruplets:
FOR I = 0 to VL step 128
    TMP1[I+31:I] := select (SRC2[I+127: I], imm8[1:0])
    TMP1[I+63: I+32] := select (SRC2[I+127: I], imm8[3:2])
    TMP1[I+95: I+64] := select (SRC2[I+127: I], imm8[5:4])
    TMP1[I+127: I+96] := select (SRC2[I+127: I], imm8[7:6])
END FOR

SAD of quadruplets:

FOR I =0 to VL step 64
    TMP_DEST[I+15:I] := ABS(SRC1[I+7: I] - TMP1[I+7: I]) +
        ABS(SRC1[I+15: I+8]- TMP1[I+15: I+8]) +

```
        ABS(SRC1[I+23: I+16]- TMP1[I+23: I+16]) +
        ABS(SRC1[I+31: I+24]- TMP1[I+31: I+24])

    TMP_DEST[I+31: I+16] := ABS(SRC1[I+7: I] - TMP1[I+15: I+8]) +
        ABS(SRC1[I+15: I+8]- TMP1[I+23: I+16]) +
        ABS(SRC1[I+23: I+16]- TMP1[I+31: I+24]) +
        ABS(SRC1[I+31: I+24]- TMP1[I+39: I+32])
    TMP_DEST[I+47: I+32] := ABS(SRC1[I+39: I+32] - TMP1[I+23: I+16]) +
        ABS(SRC1[I+47: I+40]- TMP1[I+31: I+24]) +
        ABS(SRC1[I+55: I+48]- TMP1[I+39: I+32]) +
        ABS(SRC1[I+63: I+56]- TMP1[I+47: I+40])

    TMP_DEST[I+63: I+48] := ABS(SRC1[I+39: I+32] - TMP1[I+31: I+24]) +
        ABS(SRC1[I+47: I+40] - TMP1[I+39: I+32]) +
        ABS(SRC1[I+55: I+48] - TMP1[I+47: I+40]) +
        ABS(SRC1[I+63: I+56] - TMP1[I+55: I+48])
ENDFOR

FOR j :=  0 TO KL-1
    i :=    j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] :=  TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] :=  0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] :=  0
```

Intel C/C++ Compiler Intrinsic Equivalent

VDBPSADBW __m512i _mm512_dbsad_epu8(__m512i a, __m512i b int imm8);
VDBPSADBW __m512i _mm512_mask_dbsad_epu8(__m512i s, __mmask32 m, __m512i a, __m512i b int imm8);
VDBPSADBW __m512i _mm512_maskz_dbsad_epu8(__mmask32 m, __m512i a, __m512i b int imm8);
VDBPSADBW __m256i _mm256_dbsad_epu8(__m256i a, __m256i b int imm8);
VDBPSADBW __m256i _mm256_mask_dbsad_epu8(__m256i s, __mmask16 m, __m256i a, __m256i b int imm8);
VDBPSADBW __m256i _mm256_maskz_dbsad_epu8(__mmask16 m, __m256i a, __m256i b int imm8);
VDBPSADBW __m128i _mm_dbsad_epu8(__m128i a, __m128i b int imm8);
VDBPSADBW __m128i _mm_mask_dbsad_epu8(__m128i s, __mmask8 m, __m128i a, __m128i b int imm8);
VDBPSADBW __m128i _mm_maskz_dbsad_epu8(__mmask8 m, __m128i a, __m128i b int imm8);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."

VDIVPH—Divide Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 5E /r VDIVPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Divide packed FP16 values in xmm2 by packed FP16 values in xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 5E /r VDIVPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Divide packed FP16 values in ymm2 by packed FP16 values in ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 5E /r VDIVPH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Divide packed FP16 values in zmm2 by packed FP16 values in zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction divides packed FP16 values from the first source operand by the corresponding elements in the second source operand, storing the packed FP16 result in the destination operand. The destination elements are updated according to the writemask.

Operation

VDIVPH (EVEX Encoded Versions) When SRC2 Operand is a Register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.fp16[j] := SRC1.fp16[j] / SRC2.fp16[j]
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VDIVPH (EVEX Encoded Versions) When SRC2 Operand is a Memory Source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            DEST.fp16[j] := SRC1.fp16[j] / SRC2.fp16[0]
        ELSE:
            DEST.fp16[j] := SRC1.fp16[j] / SRC2.fp16[j]
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VDIVPH __m128h _mm_div_ph (__m128h a, __m128h b);
VDIVPH __m128h _mm_mask_div_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VDIVPH __m128h _mm_maskz_div_ph (__mmask8 k, __m128h a, __m128h b);
VDIVPH __m256h _mm256_div_ph (__m256h a, __m256h b);
VDIVPH __m256h _mm256_mask_div_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VDIVPH __m256h _mm256_maskz_div_ph (__mmask16 k, __m256h a, __m256h b);
VDIVPH __m512h _mm512_div_ph (__m512h a, __m512h b);
VDIVPH __m512h _mm512_mask_div_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VDIVPH __m512h _mm512_maskz_div_ph (__mmask32 k, __m512h a, __m512h b);
VDIVPH __m512h _mm512_div_round_ph (__m512h a, __m512h b, int rounding);
VDIVPH __m512h _mm512_mask_div_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int rounding);
VDIVPH __m512h _mm512_maskz_div_round_ph (__mmask32 k, __m512h a, __m512h b, int rounding);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal, Zero.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VDIVSH—Divide Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 5E /r VDIVSH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512_FP16 OR AVX10.1	Divide low FP16 value in xmm2 by low FP16 value in xmm3/m16, and store the result in xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction divides the low FP16 value from the first source operand by the corresponding value in the second source operand, storing the FP16 result in the destination operand. Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

VDIVSH (EVEX Encoded Versions)

```
IF EVEX.b = 1 and SRC2 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    DEST.fp16[0] := SRC1.fp16[0] / SRC2.fp16[0]
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else dest.fp16[0] remains unchanged

DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VDIVSH __m128h _mm_div_round_sh (__m128h a, __m128h b, int rounding);
VDIVSH __m128h _mm_mask_div_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int rounding);
VDIVSH __m128h _mm_maskz_div_round_sh (__mmask8 k, __m128h a, __m128h b, int rounding);
VDIVSH __m128h _mm_div_sh (__m128h a, __m128h b);
VDIVSH __m128h _mm_mask_div_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VDIVSH __m128h _mm_maskz_div_sh (__mmask8 k, __m128h a, __m128h b);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal, Zero.

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VDPBF16PS—Dot Product of BF16 Pairs Accumulated Into Packed Single Precision

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 52 /r VDPBF16PS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	(AVX512_BF16 AND AVX512VL) OR AVX10.1	Multiply BF16 pairs from xmm2 and xmm3/m128, and accumulate the resulting packed single precision results in xmm1 with writemask k1.
EVEX.256.F3.0F38.W0 52 /r VDPBF16PS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	(AVX512_BF16 AND AVX512VL) OR AVX10.1	Multiply BF16 pairs from ymm2 and ymm3/m256, and accumulate the resulting packed single precision results in ymm1 with writemask k1.
EVEX.512.F3.0F38.W0 52 /r VDPBF16PS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	(AVX512_BF16 AND AVX512F) OR AVX10.1	Multiply BF16 pairs from zmm2 and zmm3/m512, and accumulate the resulting packed single precision results in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a SIMD dot-product of two BF16 pairs and accumulates into a packed single precision register.

"Round to nearest even" rounding mode is used when doing each accumulation of the FMA. Output denormals are always flushed to zero and input denormals are always treated as zero. MXCSR is not consulted nor updated.

NaN propagation priorities are described in Table 5-4.

Table 5-4. NaN Propagation Priorities

NaN Priority	Description	Comments
1	src1 low is NaN	Lower part has priority over upper part, i.e., it overrides the upper part.
2	src2 low is NaN	
3	src1 high is NaN	Upper part may be overridden if lower has NaN.
4	src2 high is NaN	
5	srcdest is NaN	Dest is propagated if no NaN is encountered by src2.

Operation

```
Define make_fp32(x):
    // The x parameter is bfloat16. Pack it in to upper 16b of a dword. The bit pattern is a legal fp32 value. Return that bit pattern.
    dword := 0
    dword[31:16] := x
    RETURN dword
```

VDPBF16PS srcdest, src1, src2

```
VL = (128, 256, 512)
KL = VL/32

origdest := srcdest
FOR i := 0 to KL-1:
    IF k1[ i ] or *no writemask*:
        IF src2 is memory and evex.b == 1:
            t := src2.dword[0]
        ELSE:
            t := src2.dword[ i ]

        // FP32 FMA with daz in, ftz out and RNE rounding. MXCSR neither consulted nor updated.

        srcdest.fp32[ i ] += make_fp32(src1.bfloat16[2*i+1]) * make_fp32(t.bfloat[1])
        srcdest.fp32[ i ] += make_fp32(src1.bfloat16[2*i+0]) * make_fp32(t.bfloat[0])

    ELSE IF *zeroing*:
        srcdest.dword[ i ] := 0
    ELSE: // merge masking, dest element unchanged
        srcdest.dword[ i ] := origdest.dword[ i ]

srcdest[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VDPBF16PS __m128 _mm_dpbf16_ps(__m128, __m128bh, __m128bh);
VDPBF16PS __m128 _mm_mask_dpbf16_ps(__m128, __mmask8, __m128bh, __m128bh);
VDPBF16PS __m128 _mm_maskz_dpbf16_ps(__mmask8, __m128, __m128bh, __m128bh);
VDPBF16PS __m256 _mm256_dpbf16_ps(__m256, __m256bh, __m256bh);
VDPBF16PS __m256 _mm256_mask_dpbf16_ps(__m256, __mmask8, __m256bh, __m256bh);
VDPBF16PS __m256 _mm256_maskz_dpbf16_ps(__mmask8, __m256, __m256bh, __m256bh);
VDPBF16PS __m512 _mm512_dpbf16_ps(__m512, __m512bh, __m512bh);
VDPBF16PS __m512 _mm512_mask_dpbf16_ps(__m512, __mmask16, __m512bh, __m512bh);
VDPBF16PS __m512 _mm512_maskz_dpbf16_ps(__mmask16, __m512, __m512bh, __m512bh);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, "Type E4 Class Exception Conditions."

VERR/VERW—Verify a Segment for Reading or Writing

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 00 /4 VERR r/m16	M	Valid	Valid	Set ZF=1 if segment specified with r/m16 can be read.
0F 00 /5 VERW r/m16	M	Valid	Valid	Set ZF=1 if segment specified with r/m16 can be written.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

Description

Verifies whether the code or data segment specified with the source operand is readable (VERR) or writable (VERW) from the current privilege level (CPL). The source operand is a 16-bit register or a memory location that contains the segment selector for the segment to be verified. If the segment is accessible and readable (VERR) or writable (VERW), the ZF flag is set; otherwise, the ZF flag is cleared. Code segments are never verified as writable. This check cannot be performed on system segments.

To set the ZF flag, the following conditions must be met:

- The segment selector is not NULL.
- The selector must denote a descriptor within the bounds of the descriptor table (GDT or LDT).
- The selector must denote the descriptor of a code or data segment (not that of a system segment or gate).
- For the VERR instruction, the segment must be readable.
- For the VERW instruction, the segment must be a writable data segment.
- If the segment is not a conforming code segment, the segment's DPL must be greater than or equal to (have less or the same privilege as) both the CPL and the segment selector's RPL.

The validation performed is the same as is performed when a segment selector is loaded into the DS, ES, FS, or GS register, and the indicated access (read or write) is performed. The segment selector's value cannot result in a protection exception, enabling the software to anticipate possible segment access problems.

This instruction's operation is the same in non-64-bit modes and 64-bit mode. The operand size is fixed at 16 bits.

Operation

```
IF SRC(Offset) > (GDTR(Limit) or (LDTR(Limit))
    THEN ZF := 0; FI;

Read segment descriptor;

IF SegmentDescriptor(DescriptorType) = 0 (* System segment *)
or (SegmentDescriptor(Type) ≠ conforming code segment)
and (CPL > DPL) or (RPL > DPL)
    THEN
        ZF := 0;
    ELSE
        IF ((Instruction = VERR) and (Segment readable))
        or ((Instruction = VERW) and (Segment writable))
            THEN
                ZF := 1;
            ELSE
                ZF := 0;
```

FI;
FI;

Flags Affected

The ZF flag is set to 1 if the segment is accessible and readable (VERR) or writable (VERW); otherwise, it is set to 0.

Protected Mode Exceptions

The only exceptions generated for these instructions are those related to illegal addressing of the source operand.

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	The VERR and VERW instructions are not recognized in real-address mode. If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD	The VERR and VERW instructions are not recognized in virtual-8086 mode. If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

VEXPANDPD—Load Sparse Packed Double Precision Floating-Point Values From Dense Memory

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 88 /r VEXPANDPD xmm1 {k1}{z}, xmm2/m128	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Expand packed double precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F38.W1 88 /r VEXPANDPD ymm1 {k1}{z}, ymm2/m256	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Expand packed double precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F38.W1 88 /r VEXPANDPD zmm1 {k1}{z}, zmm2/m512	A	V/V	AVX512F OR AVX10.1	Expand packed double precision floating-point values from zmm2/m512 to zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Expand (load) up to 8/4/2, contiguous, double precision floating-point values of the input vector in the source operand (the second operand) to sparse elements in the destination operand (the first operand) selected by the writemask k1.

The destination operand is a ZMM/YMM/XMM register, the source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The input vector starts from the lowest element in the source operand. The writemask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VEXPANDPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
k := 0
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+63:i] := SRC[k+63:k];
            k := k + 64
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    THEN DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VEXPANDPD __m512d _mm512_mask_expand_pd( __m512d s, __mmask8 k, __m512d a);
VEXPANDPD __m512d _mm512_maskz_expand_pd( __mmask8 k, __m512d a);
VEXPANDPD __m512d _mm512_mask_expandloadu_pd( __m512d s, __mmask8 k, void * a);
VEXPANDPD __m512d _mm512_maskz_expandloadu_pd( __mmask8 k, void * a);
VEXPANDPD __m256d _mm256_mask_expand_pd( __m256d s, __mmask8 k, __m256d a);
VEXPANDPD __m256d _mm256_maskz_expand_pd( __mmask8 k, __m256d a);
VEXPANDPD __m256d _mm256_mask_expandloadu_pd( __m256d s, __mmask8 k, void * a);
VEXPANDPD __m256d _mm256_maskz_expandloadu_pd( __mmask8 k, void * a);
VEXPANDPD __m128d _mm_mask_expand_pd( __m128d s, __mmask8 k, __m128d a);
VEXPANDPD __m128d _mm_maskz_expand_pd( __mmask8 k, __m128d a);
VEXPANDPD __m128d _mm_mask_expandloadu_pd( __m128d s, __mmask8 k, void * a);
VEXPANDPD __m128d _mm_maskz_expandloadu_pd( __mmask8 k, void * a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4.nb in Table 2-51, "Type E4 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VEXPANDPS—Load Sparse Packed Single Precision Floating-Point Values From Dense Memory

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 88 /r VEXPANDPS xmm1 {k1}{z}, xmm2/m128	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Expand packed single precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F38.W0 88 /r VEXPANDPS ymm1 {k1}{z}, ymm2/m256	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Expand packed single precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F38.W0 88 /r VEXPANDPS zmm1 {k1}{z}, zmm2/m512	A	V/V	AVX512F OR AVX10.1	Expand packed single precision floating-point values from zmm2/m512 to zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Expand (load) up to 16/8/4, contiguous, single precision floating-point values of the input vector in the source operand (the second operand) to sparse elements of the destination operand (the first operand) selected by the writemask k1.

The destination operand is a ZMM/YMM/XMM register, the source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The input vector starts from the lowest element in the source operand. The writemask k1 selects the destination elements (a partial vector or sparse elements if less than 16 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VEXPANDPS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
k := 0
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+31:i] := SRC[k+31:k];
            k := k + 32
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VEXPANDPS __m512 _mm512_mask_expand_ps( __m512 s, __mmask16 k, __m512 a);
VEXPANDPS __m512 _mm512_maskz_expand_ps( __mmask16 k, __m512 a);
VEXPANDPS __m512 _mm512_mask_expandloadu_ps( __m512 s, __mmask16 k, void * a);
VEXPANDPS __m512 _mm512_maskz_expandloadu_ps( __mmask16 k, void * a);
VEXPANDPD __m256 _mm256_mask_expand_ps( __m256 s, __mmask8 k, __m256 a);
VEXPANDPD __m256 _mm256_maskz_expand_ps( __mmask8 k, __m256 a);
VEXPANDPD __m256 _mm256_mask_expandloadu_ps( __m256 s, __mmask8 k, void * a);
VEXPANDPD __m256 _mm256_maskz_expandloadu_ps( __mmask8 k, void * a);
VEXPANDPD __m128 _mm_mask_expand_ps( __m128 s, __mmask8 k, __m128 a);
VEXPANDPD __m128 _mm_maskz_expand_ps( __mmask8 k, __m128 a);
VEXPANDPD __m128 _mm_mask_expandloadu_ps( __m128 s, __mmask8 k, void * a);
VEXPANDPD __m128 _mm_maskz_expandloadu_ps( __mmask8 k, void * a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4.nb in Table 2-51, "Type E4 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VEXTRACTF128/VEXTRACTF32x4/VEXTRACTF64x2/VEXTRACTF32x8/VEXTRACTF64x4— Extract Packed Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 19 /r ib VEXTRACTF128 xmm1/m128, ymm2, imm8	A	V/V	AVX	Extract 128 bits of packed floating-point values from ymm2 and store results in xmm1/m128.
EVEX.256.66.0F3A.W0 19 /r ib VEXTRACTF32X4 xmm1/m128 {k1}{z}, ymm2, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Extract 128 bits of packed single precision floating-point values from ymm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W0 19 /r ib VEXTRACTF32x4 xmm1/m128 {k1}{z}, zmm2, imm8	C	V/V	AVX512F OR AVX10.1	Extract 128 bits of packed single precision floating-point values from zmm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.256.66.0F3A.W1 19 /r ib VEXTRACTF64X2 xmm1/m128 {k1}{z}, ymm2, imm8	B	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Extract 128 bits of packed double precision floating-point values from ymm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W1 19 /r ib VEXTRACTF64X2 xmm1/m128 {k1}{z}, zmm2, imm8	B	V/V	AVX512DQ OR AVX10.1	Extract 128 bits of packed double precision floating-point values from zmm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W0 1B /r ib VEXTRACTF32X8 ymm1/m256 {k1}{z}, zmm2, imm8	D	V/V	AVX512DQ OR AVX10.1	Extract 256 bits of packed single precision floating-point values from zmm2 and store results in ymm1/m256 subject to writemask k1.
EVEX.512.66.0F3A.W1 1B /r ib VEXTRACTF64x4 ymm1/m256 {k1}{z}, zmm2, imm8	C	V/V	AVX512F OR AVX10.1	Extract 256 bits of packed double precision floating-point values from zmm2 and store results in ymm1/m256 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
B	Tuple2	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
C	Tuple4	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
D	Tuple8	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A

Description

VEXTRACTF128/VEXTRACTF32x4 and VEXTRACTF64x2 extract 128-bits of single precision floating-point values from the source operand (the second operand) and store to the low 128-bit of the destination operand (the first operand). The 128-bit data extraction occurs at an 128-bit granular offset specified by imm8[0] (256-bit) or imm8[1:0] as the multiply factor. The destination may be either a vector register or an 128-bit memory location.

VEXTRACTF32x4: The low 128-bit of the destination operand is updated at 32-bit granularity according to the writemask.

VEXTRACTF32x8 and VEXTRACTF64x4 extract 256-bits of double precision floating-point values from the source operand (second operand) and store to the low 256-bit of the destination operand (the first operand). The 256-bit data extraction occurs at an 256-bit granular offset specified by imm8[0] (256-bit) or imm8[0] as the multiply factor The destination may be either a vector register or a 256-bit memory location.

VEXTRACTF64x4: The low 256-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The high 6 bits of the immediate are ignored.

If VEXTRACTF128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

Operation

VEXTRACTF32x4 (EVEX Encoded Versions) When Destination is a Register

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] := SRC1[127:0]
        1: TMP_DEST[127:0] := SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] := SRC1[127:0]
        01: TMP_DEST[127:0] := SRC1[255:128]
        10: TMP_DEST[127:0] := SRC1[383:256]
        11: TMP_DEST[127:0] := SRC1[511:384]
    ESAC.
FI;
FOR j := 0 TO 3
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:128] := 0
```

VEXTRACTF32x4 (EVEX Encoded Versions) When Destination is Memory

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] := SRC1[127:0]
        1: TMP_DEST[127:0] := SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] := SRC1[127:0]
        01: TMP_DEST[127:0] := SRC1[255:128]
        10: TMP_DEST[127:0] := SRC1[383:256]
        11: TMP_DEST[127:0] := SRC1[511:384]
    ESAC.
FI;

FOR j := 0 TO 3
    i := j * 32
    IF k1[j] OR *no writemask*
```

```
            THEN DEST[i+31:i] := TMP_DEST[i+31:i]
            ELSE *DEST[i+31:i] remains unchanged*           ; merging-masking
        FI;
ENDFOR
```

VEXTRACTF64x2 (EVEX Encoded Versions) When Destination is a Register

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] := SRC1[127:0]
        1: TMP_DEST[127:0] := SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] := SRC1[127:0]
        01: TMP_DEST[127:0] := SRC1[255:128]
        10: TMP_DEST[127:0] := SRC1[383:256]
        11: TMP_DEST[127:0] := SRC1[511:384]
    ESAC.
FI;

FOR j := 0 TO 1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:128] := 0
```

VEXTRACTF64x2 (EVEX Encoded Versions) When Destination is Memory

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] := SRC1[127:0]
        1: TMP_DEST[127:0] := SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] := SRC1[127:0]
        01: TMP_DEST[127:0] := SRC1[255:128]
        10: TMP_DEST[127:0] := SRC1[383:256]
        11: TMP_DEST[127:0] := SRC1[511:384]
    ESAC.
FI;

FOR j := 0 TO 1
```

```
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE *DEST[i+63:i] remains unchanged*          ; merging-masking
    FI;
ENDFOR
```

VEXTRACTF32x8 (EVEX.U1.512 Encoded Version) When Destination is a Register

```
VL = 512
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC1[255:0]
    1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.

FOR j := 0 TO 7
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:256] := 0
```

VEXTRACTF32x8 (EVEX.U1.512 Encoded Version) When Destination is Memory

```
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC1[255:0]
    1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.

FOR j := 0 TO 7
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*          ; merging-masking
    FI;
ENDFOR
```

VEXTRACTF64x4 (EVEX.512 Encoded Version) When Destination is a Register

```
VL = 512
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC1[255:0]
    1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.

FOR j := 0 TO 3
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
```

```
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:256] := 0
```

VEXTRACTF64x4 (EVEX.512 Encoded Version) When Destination is Memory

```
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC1[255:0]
    1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.

FOR j := 0 TO 3
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE    ; merging-masking
            *DEST[i+63:i] remains unchanged*
    FI;
ENDFOR
```

VEXTRACTF128 (Memory Destination Form)

```
CASE (imm8[0]) OF
    0: DEST[127:0] := SRC1[127:0]
    1: DEST[127:0] := SRC1[255:128]
ESAC.
```

VEXTRACTF128 (Register Destination Form)

```
CASE (imm8[0]) OF
    0: DEST[127:0] := SRC1[127:0]
    1: DEST[127:0] := SRC1[255:128]
ESAC.
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VEXTRACTF32x4 __m128 _mm512_extractf32x4_ps(__m512 a, const int nidx);
VEXTRACTF32x4 __m128 _mm512_mask_extractf32x4_ps(__m128 s, __mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x4 __m128 _mm512_maskz_extractf32x4_ps( __mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x4 __m128 _mm256_extractf32x4_ps(__m256 a, const int nidx);
VEXTRACTF32x4 __m128 _mm256_mask_extractf32x4_ps(__m128 s, __mmask8 k, __m256 a, const int nidx);
VEXTRACTF32x4 __m128 _mm256_maskz_extractf32x4_ps( __mmask8 k, __m256 a, const int nidx);
VEXTRACTF32x8 __m256 _mm512_extractf32x8_ps(__m512 a, const int nidx);
VEXTRACTF32x8 __m256 _mm512_mask_extractf32x8_ps(__m256 s, __mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x8 __m256 _mm512_maskz_extractf32x8_ps( __mmask8 k, __m512 a, const int nidx);
VEXTRACTF64x2 __m128d _mm512_extractf64x2_pd(__m512d a, const int nidx);
VEXTRACTF64x2 __m128d _mm512_mask_extractf64x2_pd(__m128d s, __mmask8 k, __m512d a, const int nidx);
VEXTRACTF64x2 __m128d _mm512_maskz_extractf64x2_pd( __mmask8 k, __m512d a, const int nidx);
VEXTRACTF64x2 __m128d _mm256_extractf64x2_pd(__m256d a, const int nidx);
VEXTRACTF64x2 __m128d _mm256_mask_extractf64x2_pd(__m128d s, __mmask8 k, __m256d a, const int nidx);
VEXTRACTF64x2 __m128d _mm256_maskz_extractf64x2_pd( __mmask8 k, __m256d a, const int nidx);
VEXTRACTF64x4 __m256d _mm512_extractf64x4_pd( __m512d a, const int nidx);
```

VEXTRACTF64x4 __m256d _mm512_mask_extractf64x4_pd(__m256d s, __mmask8 k, __m512d a, const int nidx);
VEXTRACTF64x4 __m256d _mm512_maskz_extractf64x4_pd(__mmask8 k, __m512d a, const int nidx);
VEXTRACTF128 __m128 _mm256_extractf128_ps (__m256 a, int offset);
VEXTRACTF128 __m128d _mm256_extractf128_pd (__m256d a, int offset);
VEXTRACTF128 __m128i_mm256_extractf128_si256(__m256i a, int offset);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instructions, see Table 2-23, "Type 6 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-56, "Type E6NF Class Exception Conditions."

Additionally:

#UD	IF VEX.L = 0.
#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

VEXTRACTI128/VEXTRACTI32x4/VEXTRACTI64x2/VEXTRACTI32x8/VEXTRACTI64x4—Extract Packed Integer Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	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/m128.
EVEX.256.66.0F3A.W0 39 /r ib VEXTRACTI32X4 xmm1/m128 {k1}{z}, ymm2, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Extract 128 bits of double-word integer values from ymm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W0 39 /r ib VEXTRACTI32x4 xmm1/m128 {k1}{z}, zmm2, imm8	C	V/V	AVX512F OR AVX10.1	Extract 128 bits of double-word integer values from zmm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.256.66.0F3A.W1 39 /r ib VEXTRACTI64X2 xmm1/m128 {k1}{z}, ymm2, imm8	B	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Extract 128 bits of quad-word integer values from ymm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W1 39 /r ib VEXTRACTI64X2 xmm1/m128 {k1}{z}, zmm2, imm8	B	V/V	AVX512DQ OR AVX10.1	Extract 128 bits of quad-word integer values from zmm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W0 3B /r ib VEXTRACTI32X8 ymm1/m256 {k1}{z}, zmm2, imm8	D	V/V	AVX512DQ OR AVX10.1	Extract 256 bits of double-word integer values from zmm2 and store results in ymm1/m256 subject to writemask k1.
EVEX.512.66.0F3A.W1 3B /r ib VEXTRACTI64x4 ymm1/m256 {k1}{z}, zmm2, imm8	C	V/V	AVX512F OR AVX10.1	Extract 256 bits of quad-word integer values from zmm2 and store results in ymm1/m256 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
B	Tuple2	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
C	Tuple4	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
D	Tuple8	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A

Description

VEXTRACTI128/VEXTRACTI32x4 and VEXTRACTI64x2 extract 128-bits of doubleword integer values from the source operand (the second operand) and store to the low 128-bit of the destination operand (the first operand). The 128-bit data extraction occurs at an 128-bit granular offset specified by imm8[0] (256-bit) or imm8[1:0] as the multiply factor. The destination may be either a vector register or an 128-bit memory location.

VEXTRACTI32x4: The low 128-bit of the destination operand is updated at 32-bit granularity according to the writemask.

VEXTRACTI64x2: The low 128-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEXTRACTI32x8 and VEXTRACTI64x4 extract 256-bits of quadword integer values from the source operand (the second operand) and store to the low 256-bit of the destination operand (the first operand). The 256-bit data extraction occurs at an 256-bit granular offset specified by imm8[0] (256-bit) or imm8[0] as the multiply factor The destination may be either a vector register or a 256-bit memory location.

VEXTRACTI32x8: The low 256-bit of the destination operand is updated at 32-bit granularity according to the writemask.

VEXTRACTI64x4: The low 256-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The high 7 bits (6 bits in EVEX.512) of the immediate are ignored.

If VEXTRACTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

Operation

VEXTRACTI32x4 (EVEX encoded versions) when destination is a register

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] := SRC1[127:0]
        1: TMP_DEST[127:0] := SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] := SRC1[127:0]
        01: TMP_DEST[127:0] := SRC1[255:128]
        10: TMP_DEST[127:0] := SRC1[383:256]
        11: TMP_DEST[127:0] := SRC1[511:384]
    ESAC.
FI;
FOR j := 0 TO 3
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:128] := 0
```

VEXTRACTI32x4 (EVEX encoded versions) when destination is memory

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] := SRC1[127:0]
        1: TMP_DEST[127:0] := SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] := SRC1[127:0]
        01: TMP_DEST[127:0] := SRC1[255:128]
        10: TMP_DEST[127:0] := SRC1[383:256]
        11: TMP_DEST[127:0] := SRC1[511:384]
    ESAC.
```

```
FI;

FOR j := 0 TO 3
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*          ; merging-masking
    FI;
ENDFOR
```

VEXTRACTI64x2 (EVEX encoded versions) when destination is a register

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] := SRC1[127:0]
        1: TMP_DEST[127:0] := SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] := SRC1[127:0]
        01: TMP_DEST[127:0] := SRC1[255:128]
        10: TMP_DEST[127:0] := SRC1[383:256]
        11: TMP_DEST[127:0] := SRC1[511:384]
    ESAC.
FI;

FOR j := 0 TO 1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:128] := 0
```

VEXTRACTI64x2 (EVEX encoded versions) when destination is memory

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] := SRC1[127:0]
        1: TMP_DEST[127:0] := SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] := SRC1[127:0]
        01: TMP_DEST[127:0] := SRC1[255:128]
        10: TMP_DEST[127:0] := SRC1[383:256]
```

```
        11: TMP_DEST[127:0] := SRC1[511:384]
    ESAC.
FI;

FOR j := 0 TO 1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE *DEST[i+63:i] remains unchanged*          ; merging-masking
    FI;
ENDFOR
```

VEXTRACTI32x8 (EVEX.U1.512 encoded version) when destination is a register

```
VL = 512
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC1[255:0]
    1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.

FOR j := 0 TO 7
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:256] := 0
```

VEXTRACTI32x8 (EVEX.U1.512 encoded version) when destination is memory

```
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC1[255:0]
    1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.

FOR j := 0 TO 7
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*          ; merging-masking
    FI;
ENDFOR
```

VEXTRACTI64x4 (EVEX.512 encoded version) when destination is a register

```
VL = 512
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC1[255:0]
    1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.

FOR j := 0 TO 3
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:256] := 0
```

VEXTRACTI64x4 (EVEX.512 encoded version) when destination is memory

```
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC1[255:0]
    1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.
FOR j := 0 TO 3
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE *DEST[i+63:i] remains unchanged*       ; merging-masking
    FI;
ENDFOR
```

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[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VEXTRACTI32x4 __m128i _mm512_extracti32x4_epi32(__m512i a, const int nidx);
VEXTRACTI32x4 __m128i _mm512_mask_extracti32x4_epi32(__m128i s, __mmask8 k, __m512i a, const int nidx);
VEXTRACTI32x4 __m128i _mm512_maskz_extracti32x4_epi32(__mmask8 k, __m512i a, const int nidx);
VEXTRACTI32x4 __m128i _mm256_extracti32x4_epi32(__m256i a, const int nidx);
VEXTRACTI32x4 __m128i _mm256_mask_extracti32x4_epi32(__m128i s, __mmask8 k, __m256i a, const int nidx);
VEXTRACTI32x4 __m128i _mm256_maskz_extracti32x4_epi32(__mmask8 k, __m256i a, const int nidx);
VEXTRACTI32x8 __m256i _mm512_extracti32x8_epi32(__m512i a, const int nidx);
VEXTRACTI32x8 __m256i _mm512_mask_extracti32x8_epi32(__m256i s, __mmask8 k, __m512i a, const int nidx);
VEXTRACTI32x8 __m256i _mm512_maskz_extracti32x8_epi32(__mmask8 k, __m512i a, const int nidx);
VEXTRACTI64x2 __m128i _mm512_extracti64x2_epi64(__m512i a, const int nidx);
VEXTRACTI64x2 __m128i _mm512_mask_extracti64x2_epi64(__m128i s, __mmask8 k, __m512i a, const int nidx);
VEXTRACTI64x2 __m128i _mm512_maskz_extracti64x2_epi64(__mmask8 k, __m512i a, const int nidx);
VEXTRACTI64x2 __m128i _mm256_extracti64x2_epi64(__m256i a, const int nidx);
VEXTRACTI64x2 __m128i _mm256_mask_extracti64x2_epi64(__m128i s, __mmask8 k, __m256i a, const int nidx);
VEXTRACTI64x2 __m128i _mm256_maskz_extracti64x2_epi64(__mmask8 k, __m256i a, const int nidx);
VEXTRACTI64x4 __m256i _mm512_extracti64x4_epi64(__m512i a, const int nidx);
VEXTRACTI64x4 __m256i _mm512_mask_extracti64x4_epi64(__m256i s, __mmask8 k, __m512i a, const int nidx);
VEXTRACTI64x4 __m256i _mm512_maskz_extracti64x4_epi64(__mmask8 k, __m512i a, const int nidx);
VEXTRACTI128 __m128i _mm256_extracti128_si256(__m256i a, int offset);

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instructions, see Table 2-23, "Type 6 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-56, "Type E6NF Class Exception Conditions."
Additionally:

#UD	IF VEX.L = 0.
#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

VFCMADDCPH/VFMADDCPH—Complex Multiply and Accumulate FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.MAP6.W0 56 /r VFCMADDCPH xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Complex multiply a pair of FP16 values from xmm2 and xmm3/m128/m32bcst, add to xmm1 and store the result in xmm1 subject to writemask k1.
EVEX.256.F2.MAP6.W0 56 /r VFCMADDCPH ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Complex multiply a pair of FP16 values from ymm2 and ymm3/m256/m32bcst, add to ymm1 and store the result in ymm1 subject to writemask k1.
EVEX.512.F2.MAP6.W0 56 /r VFCMADDCPH zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Complex multiply a pair of FP16 values from zmm2 and zmm3/m512/m32bcst, add to zmm1 and store the result in zmm1 subject to writemask k1.
EVEX.128.F3.MAP6.W0 56 /r VFMADDCPH xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Complex multiply a pair of FP16 values from xmm2 and the complex conjugate of xmm3/m128/m32bcst, add to xmm1 and store the result in xmm1 subject to writemask k1.
EVEX.256.F3.MAP6.W0 56 /r VFMADDCPH ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Complex multiply a pair of FP16 values from ymm2 and the complex conjugate of ymm3/m256/m32bcst, add to ymm1 and store the result in ymm1 subject to writemask k1.
EVEX.512.F3.MAP6.W0 56 /r VFMADDCPH zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Complex multiply a pair of FP16 values from zmm2 and the complex conjugate of zmm3/m512/m32bcst, add to zmm1 and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a complex multiply and accumulate operation. There are normal and complex conjugate forms of the operation.

The broadcasting and masking for this operation is done on 32-bit quantities representing a pair of FP16 values.

Rounding is performed at every FMA (fused multiply and add) boundary. Execution occurs as if all MXCSR exceptions are masked. MXCSR status bits are updated to reflect exceptional conditions.

Operation

VFCMADDCPH dest{k1}, src1, src2 (AVX512)

```
VL = 128, 256, 512
KL := VL / 32

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF broadcasting and src2 is memory:
            tsrc2.fp16[2*i+0] := src2.fp16[0]
            tsrc2.fp16[2*i+1] := src2.fp16[1]
        ELSE:
            tsrc2.fp16[2*i+0] := src2.fp16[2*i+0]
            tsrc2.fp16[2*i+1] := src2.fp16[2*i+1]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        tmp[2*i+0] := dest.fp16[2*i+0] + src1.fp16[2*i+0] * tsrc2.fp16[2*i+0]
        tmp[2*i+1] := dest.fp16[2*i+1] + src1.fp16[2*i+1] * tsrc2.fp16[2*i+0]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        // conjugate version subtracts odd final term
        dest.fp16[2*i+0] := tmp[2*i+0] + src1.fp16[2*i+1] * tsrc2.fp16[2*i+1]
        dest.fp16[2*i+1] := tmp[2*i+1] - src1.fp16[2*i+0] * tsrc2.fp16[2*i+1]
    ELSE IF *zeroing*:
        dest.fp16[2*i+0] := 0
        dest.fp16[2*i+1] := 0

DEST[MAXVL-1:VL] := 0
```

VFMADDCPH dest{k1}, src1, src2 (AVX512)

```
VL = 128, 256, 512
KL := VL / 32

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF broadcasting and src2 is memory:
            tsrc2.fp16[2*i+0] := src2.fp16[0]
            tsrc2.fp16[2*i+1] := src2.fp16[1]
        ELSE:
            tsrc2.fp16[2*i+0] := src2.fp16[2*i+0]
            tsrc2.fp16[2*i+1] := src2.fp16[2*i+1]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        tmp[2*i+0] := dest.fp16[2*i+0] + src1.fp16[2*i+0] * tsrc2.fp16[2*i+0]
        tmp[2*i+1] := dest.fp16[2*i+1] + src1.fp16[2*i+1] * tsrc2.fp16[2*i+0]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        // non-conjugate version subtracts even term
        dest.fp16[2*i+0] := tmp[2*i+0] - src1.fp16[2*i+1] * tsrc2.fp16[2*i+1]
        dest.fp16[2*i+1] := tmp[2*i+1] + src1.fp16[2*i+0] * tsrc2.fp16[2*i+1]
    ELSE IF *zeroing*:
```

```
        dest.fp16[2*i+0] := 0
        dest.fp16[2*i+1] := 0

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFCMADDCPH __m128h _mm_fcmadd_pch (__m128h a, __m128h b, __m128h c);
VFCMADDCPH __m128h _mm_mask_fcmadd_pch (__m128h a, __mmask8 k, __m128h b, __m128h c);
VFCMADDCPH __m128h _mm_mask3_fcmadd_pch (__m128h a, __m128h b, __m128h c, __mmask8 k);
VFCMADDCPH __m128h _mm_maskz_fcmadd_pch (__mmask8 k, __m128h a, __m128h b, __m128h c);
VFCMADDCPH __m256h _mm256_fcmadd_pch (__m256h a, __m256h b, __m256h c);
VFCMADDCPH __m256h _mm256_mask_fcmadd_pch (__m256h a, __mmask8 k, __m256h b, __m256h c);
VFCMADDCPH __m256h _mm256_mask3_fcmadd_pch (__m256h a, __m256h b, __m256h c, __mmask8 k);
VFCMADDCPH __m256h _mm256_maskz_fcmadd_pch (__mmask8 k, __m256h a, __m256h b, __m256h c);
VFCMADDCPH __m512h _mm512_fcmadd_pch (__m512h a, __m512h b, __m512h c);
VFCMADDCPH __m512h _mm512_mask_fcmadd_pch (__m512h a, __mmask16 k, __m512h b, __m512h c);
VFCMADDCPH __m512h _mm512_mask3_fcmadd_pch (__m512h a, __m512h b, __m512h c, __mmask16 k);
VFCMADDCPH __m512h _mm512_maskz_fcmadd_pch (__mmask16 k, __m512h a, __m512h b, __m512h c);
VFCMADDCPH __m512h _mm512_fcmadd_round_pch (__m512h a, __m512h b, __m512h c, const int rounding);
VFCMADDCPH __m512h _mm512_mask_fcmadd_round_pch (__m512h a, __mmask16 k, __m512h b, __m512h c, const int rounding);
VFCMADDCPH __m512h _mm512_mask3_fcmadd_round_pch (__m512h a, __m512h b, __m512h c, __mmask16 k, const int rounding);
VFCMADDCPH __m512h _mm512_maskz_fcmadd_round_pch (__mmask16 k, __m512h a, __m512h b, __m512h c, const int rounding);

VFMADDCPH __m128h _mm_fmadd_pch (__m128h a, __m128h b, __m128h c);
VFMADDCPH __m128h _mm_mask_fmadd_pch (__m128h a, __mmask8 k, __m128h b, __m128h c);
VFMADDCPH __m128h _mm_mask3_fmadd_pch (__m128h a, __m128h b, __m128h c, __mmask8 k);
VFMADDCPH __m128h _mm_maskz_fmadd_pch (__mmask8 k, __m128h a, __m128h b, __m128h c);
VFMADDCPH __m256h _mm256_fmadd_pch (__m256h a, __m256h b, __m256h c);
VFMADDCPH __m256h _mm256_mask_fmadd_pch (__m256h a, __mmask8 k, __m256h b, __m256h c);
VFMADDCPH __m256h _mm256_mask3_fmadd_pch (__m256h a, __m256h b, __m256h c, __mmask8 k);
VFMADDCPH __m256h _mm256_maskz_fmadd_pch (__mmask8 k, __m256h a, __m256h b, __m256h c);
VFMADDCPH __m512h _mm512_fmadd_pch (__m512h a, __m512h b, __m512h c);
VFMADDCPH __m512h _mm512_mask_fmadd_pch (__m512h a, __mmask16 k, __m512h b, __m512h c);
VFMADDCPH __m512h _mm512_mask3_fmadd_pch (__m512h a, __m512h b, __m512h c, __mmask16 k);
VFMADDCPH __m512h _mm512_maskz_fmadd_pch (__mmask16 k, __m512h a, __m512h b, __m512h c);
VFMADDCPH __m512h _mm512_fmadd_round_pch (__m512h a, __m512h b, __m512h c, const int rounding);
VFMADDCPH __m512h _mm512_mask_fmadd_round_pch (__m512h a, __mmask16 k, __m512h b, __m512h c, const int rounding);
VFMADDCPH __m512h _mm512_mask3_fmadd_round_pch (__m512h a, __m512h b, __m512h c, __mmask16 k, const int rounding);
VFMADDCPH __m512h _mm512_maskz_fmadd_round_pch (__mmask16 k, __m512h a, __m512h b, __m512h c, const int rounding);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-51, “Type E4 Class Exception Conditions.”
Additionally:

#UD	If (dest_reg == src1_reg) or (dest_reg == src2_reg).

VFCMADDCSH/VFMADDCSH—Complex Multiply and Accumulate Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F2.MAP6.W0 57 /r VFCMADDCSH xmm1{k1}{z}, xmm2, xmm3/m32 {er}	A	V/V	AVX512_FP16 OR AVX10.1	Complex multiply a pair of FP16 values from xmm2 and xmm3/m32, add to xmm1 and store the result in xmm1 subject to writemask k1. Bits 127:32 of xmm2 are copied to xmm1[127:32].
EVEX.LLIG.F3.MAP6.W0 57 /r VFMADDCSH xmm1{k1}{z}, xmm2, xmm3/m32 {er}	A	V/V	AVX512_FP16 OR AVX10.1	Complex multiply a pair of FP16 values from xmm2 and the complex conjugate of xmm3/m32, add to xmm1 and store the result in xmm1 subject to writemask k1. Bits 127:32 of xmm2 are copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a complex multiply and accumulate operation. There are normal and complex conjugate forms of the operation.

The masking for this operation is done on 32-bit quantities representing a pair of FP16 values.

Bits 127:32 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Rounding is performed at every FMA (fused multiply and add) boundary. Execution occurs as if all MXCSR exceptions are masked. MXCSR status bits are updated to reflect exceptional conditions.

Operation

VFCMADDCSH dest{k1}, src1, src2 (AVX512)

```
IF k1[0] or *no writemask*:
    tmp[0] := dest.fp16[0] + src1.fp16[0] * src2.fp16[0]
    tmp[1] := dest.fp16[1] + src1.fp16[1] * src2.fp16[0]

    // conjugate version subtracts odd final term
    dest.fp16[0] := tmp[0] + src1.fp16[1] * src2.fp16[1]
    dest.fp16[1] := tmp[1] - src1.fp16[0] * src2.fp16[1]
ELSE IF *zeroing*:
    dest.fp16[0] := 0
    dest.fp16[1] := 0

DEST[127:32] := src1[127:32] // copy upper part of src1
DEST[MAXVL-1:128] := 0
```

VFMADDCSH dest{k1}, src1, src2 (AVX512)

```
IF k1[0] or *no writemask*:
    tmp[0] := dest.fp16[0] + src1.fp16[0] * src2.fp16[0]
    tmp[1] := dest.fp16[1] + src1.fp16[1] * src2.fp16[0]

    // non-conjugate version subtracts last even term
    dest.fp16[0] := tmp[0] - src1.fp16[1] * src2.fp16[1]
    dest.fp16[1] := tmp[1] + src1.fp16[0] * src2.fp16[1]
ELSE IF *zeroing*:
    dest.fp16[0] := 0
    dest.fp16[1] := 0

DEST[127:32] := src1[127:32] // copy upper part of src1
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFCMADDCSH __m128h _mm_fcmadd_round_sch (__m128h a, __m128h b, __m128h c, const int rounding);
VFCMADDCSH __m128h _mm_mask_fcmadd_round_sch (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
VFCMADDCSH __m128h _mm_mask3_fcmadd_round_sch (__m128h a, __m128h b, __m128h c, __mmask8 k, const int rounding);
VFCMADDCSH __m128h _mm_maskz_fcmadd_round_sch (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
VFCMADDCSH __m128h _mm_fcmadd_sch (__m128h a, __m128h b, __m128h c);
VFCMADDCSH __m128h _mm_mask_fcmadd_sch (__m128h a, __mmask8 k, __m128h b, __m128h c);
VFCMADDCSH __m128h _mm_mask3_fcmadd_sch (__m128h a, __m128h b, __m128h c, __mmask8 k);
VFCMADDCSH __m128h _mm_maskz_fcmadd_sch (__mmask8 k, __m128h a, __m128h b, __m128h c);

VFMADDCSH __m128h _mm_fmadd_round_sch (__m128h a, __m128h b, __m128h c, const int rounding);
VFMADDCSH __m128h _mm_mask_fmadd_round_sch (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
VFMADDCSH __m128h _mm_mask3_fmadd_round_sch (__m128h a, __m128h b, __m128h c, __mmask8 k, const int rounding);
VFMADDCSH __m128h _mm_maskz_fmadd_round_sch (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
VFMADDCSH __m128h _mm_fmadd_sch (__m128h a, __m128h b, __m128h c);
VFMADDCSH __m128h _mm_mask_fmadd_sch (__m128h a, __mmask8 k, __m128h b, __m128h c);
VFMADDCSH __m128h _mm_mask3_fmadd_sch (__m128h a, __m128h b, __m128h c, __mmask8 k);
VFMADDCSH __m128h _mm_maskz_fmadd_sch (__mmask8 k, __m128h a, __m128h b, __m128h c);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-60, “Type E10 Class Exception Conditions.”
Additionally:

#UD	If (dest_reg == src1_reg) or (dest_reg == src2_reg).

VFCMULCPH/VFMULCPH—Complex Multiply FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.MAP6.W0 D6 /r VFCMULCPH xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Complex multiply a pair of FP16 values from xmm2 and xmm3/m128/m32bcst, and store the result in xmm1 subject to writemask k1.
EVEX.256.F2.MAP6.W0 D6 /r VFCMULCPH ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Complex multiply a pair of FP16 values from ymm2 and ymm3/m256/m32bcst, and store the result in ymm1 subject to writemask k1.
EVEX.512.F2.MAP6.W0 D6 /r VFCMULCPH zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Complex multiply a pair of FP16 values from zmm2 and zmm3/m512/m32bcst, and store the result in zmm1 subject to writemask k1.
EVEX.128.F3.MAP6.W0 D6 /r VFMULCPH xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Complex multiply a pair of FP16 values from xmm2 and the complex conjugate of xmm3/m128/m32bcst, and store the result in xmm1 subject to writemask k1.
EVEX.256.F3.MAP6.W0 D6 /r VFMULCPH ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Complex multiply a pair of FP16 values from ymm2 and the complex conjugate of ymm3/m256/m32bcst, and store the result in ymm1 subject to writemask k1.
EVEX.512.F3.MAP6.W0 D6 /r VFMULCPH zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Complex multiply a pair of FP16 values from zmm2 and the complex conjugate of zmm3/m512/m32bcst, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a complex multiply operation. There are normal and complex conjugate forms of the operation. The broadcasting and masking for this operation is done on 32-bit quantities representing a pair of FP16 values.

Rounding is performed at every FMA (fused multiply and add) boundary. Execution occurs as if all MXCSR exceptions are masked. MXCSR status bits are updated to reflect exceptional conditions.

Operation

VFCMULCPH dest{k1}, src1, src2 (AVX512)

```
VL = 128, 256 or 512
KL := VL/32

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF broadcasting and src2 is memory:
            tsrc2.fp16[2*i+0] := src2.fp16[0]
            tsrc2.fp16[2*i+1] := src2.fp16[1]
        ELSE:
            tsrc2.fp16[2*i+0] := src2.fp16[2*i+0]
            tsrc2.fp16[2*i+1] := src2.fp16[2*i+1]

FOR i := 0 to KL-1:
```

```
    IF k1[i] or *no writemask*:
        tmp.fp16[2*i+0] := src1.fp16[2*i+0] * tsrc2.fp16[2*i+0]
        tmp.fp16[2*i+1] := src1.fp16[2*i+1] * tsrc2.fp16[2*i+0]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        // conjugate version subtracts odd final term
        dest.fp16[2*i] := tmp.fp16[2*i+0] +src1.fp16[2*i+1] * tsrc2.fp16[2*i+1]
        dest.fp16[2*i+1] := tmp.fp16[2*i+1] - src1.fp16[2*i+0] * tsrc2.fp16[2*i+1]
    ELSE IF *zeroing*:
        dest.fp16[2*i+0] := 0
        dest.fp16[2*i+1] := 0

DEST[MAXVL-1:VL] := 0
```

VFMULCPH dest{k1}, src1, src2 (AVX512)

```
VL = 128, 256 or 512
KL := VL/32

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF broadcasting and src2 is memory:
            tsrc2.fp16[2*i+0] := src2.fp16[0]
            tsrc2.fp16[2*i+1] := src2.fp16[1]
        ELSE:
            tsrc2.fp16[2*i+0] := src2.fp16[2*i+0]
            tsrc2.fp16[2*i+1] := src2.fp16[2*i+1]

FOR i := 0 to kl-1:
    IF k1[i] or *no writemask*:
        tmp.fp16[2*i+0] := src1.fp16[2*i+0] * tsrc2.fp16[2*i+0]
        tmp.fp16[2*i+1] := src1.fp16[2*i+1] * tsrc2.fp16[2*i+0]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        // non-conjugate version subtracts last even term
        dest.fp16[2*i+0] := tmp.fp16[2*i+0] - src1.fp16[2*i+1] * tsrc2.fp16[2*i+1]
        dest.fp16[2*i+1] := tmp.fp16[2*i+1] + src1.fp16[2*i+0] * tsrc2.fp16[2*i+1]
    ELSE IF *zeroing*:
        dest.fp16[2*i+0] := 0
        dest.fp16[2*i+1] := 0

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFCMULCPH __m128h _mm_cmul_pch (__m128h a, __m128h b);
VFCMULCPH __m128h _mm_mask_cmul_pch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFCMULCPH __m128h _mm_maskz_cmul_pch (__mmask8 k, __m128h a, __m128h b);
VFCMULCPH __m256h _mm256_cmul_pch (__m256h a, __m256h b);
VFCMULCPH __m256h _mm256_mask_cmul_pch (__m256h src, __mmask8 k, __m256h a, __m256h b);
VFCMULCPH __m256h _mm256_maskz_cmul_pch (__mmask8 k, __m256h a, __m256h b);
VFCMULCPH __m512h _mm512_cmul_pch (__m512h a, __m512h b);
VFCMULCPH __m512h _mm512_mask_cmul_pch (__m512h src, __mmask16 k, __m512h a, __m512h b);
VFCMULCPH __m512h _mm512_maskz_cmul_pch (__mmask16 k, __m512h a, __m512h b);
VFCMULCPH __m512h _mm512_cmul_round_pch (__m512h a, __m512h b, const int rounding);
VFCMULCPH __m512h _mm512_mask_cmul_round_pch (__m512h src, __mmask16 k, __m512h a, __m512h b, const int rounding);
VFCMULCPH __m512h _mm512_maskz_cmul_round_pch (__mmask16 k, __m512h a, __m512h b, const int rounding);
VFCMULCPH __m128h _mm_fcmul_pch (__m128h a, __m128h b);
VFCMULCPH __m128h _mm_mask_fcmul_pch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFCMULCPH __m128h _mm_maskz_fcmul_pch (__mmask8 k, __m128h a, __m128h b);
VFCMULCPH __m256h _mm256_fcmul_pch (__m256h a, __m256h b);
VFCMULCPH __m256h _mm256_mask_fcmul_pch (__m256h src, __mmask8 k, __m256h a, __m256h b);
VFCMULCPH __m256h _mm256_maskz_fcmul_pch (__mmask8 k, __m256h a, __m256h b);
VFCMULCPH __m512h _mm512_fcmul_pch (__m512h a, __m512h b);
VFCMULCPH __m512h _mm512_mask_fcmul_pch (__m512h src, __mmask16 k, __m512h a, __m512h b);
VFCMULCPH __m512h _mm512_maskz_fcmul_pch (__mmask16 k, __m512h a, __m512h b);
VFCMULCPH __m512h _mm512_fcmul_round_pch (__m512h a, __m512h b, const int rounding);
VFCMULCPH __m512h _mm512_mask_fcmul_round_pch (__m512h src, __mmask16 k, __m512h a, __m512h b, const int rounding);
VFCMULCPH __m512h _mm512_maskz_fcmul_round_pch (__mmask16 k, __m512h a, __m512h b, const int rounding);

VFMULCPH __m128h _mm_fmul_pch (__m128h a, __m128h b);
VFMULCPH __m128h _mm_mask_fmul_pch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFMULCPH __m128h _mm_maskz_fmul_pch (__mmask8 k, __m128h a, __m128h b);
VFMULCPH __m256h _mm256_fmul_pch (__m256h a, __m256h b);
VFMULCPH __m256h _mm256_mask_fmul_pch (__m256h src, __mmask8 k, __m256h a, __m256h b);
VFMULCPH __m256h _mm256_maskz_fmul_pch (__mmask8 k, __m256h a, __m256h b);
VFMULCPH __m512h _mm512_fmul_pch (__m512h a, __m512h b);
VFMULCPH __m512h _mm512_mask_fmul_pch (__m512h src, __mmask16 k, __m512h a, __m512h b);
VFMULCPH __m512h _mm512_maskz_fmul_pch (__mmask16 k, __m512h a, __m512h b);
VFMULCPH __m512h _mm512_fmul_round_pch (__m512h a, __m512h b, const int rounding);
VFMULCPH __m512h _mm512_mask_fmul_round_pch (__m512h src, __mmask16 k, __m512h a, __m512h b, const int rounding);
VFMULCPH __m512h _mm512_maskz_fmul_round_pch (__mmask16 k, __m512h a, __m512h b, const int rounding);
VFMULCPH __m128h _mm_mask_mul_pch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFMULCPH __m128h _mm_maskz_mul_pch (__mmask8 k, __m128h a, __m128h b);
VFMULCPH __m128h _mm_mul_pch (__m128h a, __m128h b);
VFMULCPH __m256h _mm256_mask_mul_pch (__m256h src, __mmask8 k, __m256h a, __m256h b);
VFMULCPH __m256h _mm256_maskz_mul_pch (__mmask8 k, __m256h a, __m256h b);
VFMULCPH __m256h _mm256_mul_pch (__m256h a, __m256h b);
VFMULCPH __m512h _mm512_mask_mul_pch (__m512h src, __mmask16 k, __m512h a, __m512h b);
VFMULCPH __m512h _mm512_maskz_mul_pch (__mmask16 k, __m512h a, __m512h b);
VFMULCPH __m512h _mm512_mul_pch (__m512h a, __m512h b);
VFMULCPH __m512h _mm512_mask_mul_round_pch (__m512h src, __mmask16 k, __m512h a, __m512h b, const int rounding);
VFMULCPH __m512h _mm512_maskz_mul_round_pch (__mmask16 k, __m512h a, __m512h b, const int rounding);
VFMULCPH __m512h _mm512_mul_round_pch (__m512h a, __m512h b, const int rounding);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-51, “Type E4 Class Exception Conditions.”
Additionally:

#UD	If (dest_reg == src1_reg) or (dest_reg == src2_reg).

VFCMULCSH/VFMULCSH—Complex Multiply Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F2.MAP6.W0 D7 /r VFCMULCSH xmm1{k1}{z}, xmm2, xmm3/m32 {er}	A	V/V	AVX512_FP16 OR AVX10.1	Complex multiply a pair of FP16 values from xmm2 and xmm3/m32, and store the result in xmm1 subject to writemask k1. Bits 127:32 of xmm2 are copied to xmm1[127:32].
EVEX.LLIG.F3.MAP6.W0 D7 /r VFMULCSH xmm1{k1}{z}, xmm2, xmm3/m32 {er}	A	V/V	AVX512_FP16 OR AVX10.1	Complex multiply a pair of FP16 values from xmm2 and the complex conjugate of xmm3/m32, and store the result in xmm1 subject to writemask k1. Bits 127:32 of xmm2 are copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a complex multiply operation. There are normal and complex conjugate forms of the operation. The masking for this operation is done on 32-bit quantities representing a pair of FP16 values.

Bits 127:32 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Rounding is performed at every FMA (fused multiply and add) boundary. Execution occurs as if all MXCSR exceptions are masked. MXCSR status bits are updated to reflect exceptional conditions.

Operation

VFCMULCSH dest{k1}, src1, src2 (AVX512)

```
KL := VL / 32

IF k1[0] or *no writemask*:
    tmp.fp16[0] := src1.fp16[0] * src2.fp16[0]
    tmp.fp16[1] := src1.fp16[1] * src2.fp16[0]

    // conjugate version subtracts odd final term
    dest.fp16[0] := tmp.fp16[0] + src1.fp16[1] * src2.fp16[1]
    dest.fp16[1] := tmp.fp16[1] - src1.fp16[0] * src2.fp16[1]
ELSE IF *zeroing*:
    dest.fp16[0] := 0
    dest.fp16[1] := 0

DEST[127:32] := src1[127:32] // copy upper part of src1
DEST[MAXVL-1:128] := 0
```

VFMULCSH dest{k1}, src1, src2 (AVX512)

```
KL := VL / 32

IF k1[0] or *no writemask*:
    // non-conjugate version subtracts last even term
    tmp.fp16[0] := src1.fp16[0] * src2.fp16[0]
    tmp.fp16[1] := src1.fp16[1] * src2.fp16[0]
    dest.fp16[0] := tmp.fp16[0] - src1.fp16[1] * src2.fp16[1]
    dest.fp16[1] := tmp.fp16[1] + src1.fp16[0] * src2.fp16[1]
ELSE IF *zeroing*:
    dest.fp16[0] := 0
    dest.fp16[1] := 0

DEST[127:32] := src1[127:32] // copy upper part of src1
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFCMULCSH __m128h _mm_cmul_round_sch (__m128h a, __m128h b, const int rounding);
VFCMULCSH __m128h _mm_mask_cmul_round_sch (__m128h src, __mmask8 k, __m128h a, __m128h b, const int rounding);
VFCMULCSH __m128h _mm_maskz_cmul_round_sch (__mmask8 k, __m128h a, __m128h b, const int rounding);
VFCMULCSH __m128h _mm_cmul_sch (__m128h a, __m128h b);
VFCMULCSH __m128h _mm_mask_cmul_sch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFCMULCSH __m128h _mm_maskz_cmul_sch (__mmask8 k, __m128h a, __m128h b);
VFCMULCSH __m128h _mm_fcmul_round_sch (__m128h a, __m128h b, const int rounding);
VFCMULCSH __m128h _mm_mask_fcmul_round_sch (__m128h src, __mmask8 k, __m128h a, __m128h b, const int rounding);
VFCMULCSH __m128h _mm_maskz_fcmul_round_sch (__mmask8 k, __m128h a, __m128h b, const int rounding);
VFCMULCSH __m128h _mm_fcmul_sch (__m128h a, __m128h b);
VFCMULCSH __m128h _mm_mask_fcmul_sch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFCMULCSH __m128h _mm_maskz_fcmul_sch (__mmask8 k, __m128h a, __m128h b);

VFMULCSH __m128h _mm_fmul_round_sch (__m128h a, __m128h b, const int rounding);
VFMULCSH __m128h _mm_mask_fmul_round_sch (__m128h src, __mmask8 k, __m128h a, __m128h b, const int rounding);
VFMULCSH __m128h _mm_maskz_fmul_round_sch (__mmask8 k, __m128h a, __m128h b, const int rounding);
VFMULCSH __m128h _mm_fmul_sch (__m128h a, __m128h b);
VFMULCSH __m128h _mm_mask_fmul_sch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFMULCSH __m128h _mm_maskz_fmul_sch (__mmask8 k, __m128h a, __m128h b);
VFMULCSH __m128h _mm_mask_mul_round_sch (__m128h src, __mmask8 k, __m128h a, __m128h b, const int rounding);
VFMULCSH __m128h _mm_maskz_mul_round_sch (__mmask8 k, __m128h a, __m128h b, const int rounding);
VFMULCSH __m128h _mm_mul_round_sch (__m128h a, __m128h b, const int rounding);
VFMULCSH __m128h _mm_mask_mul_sch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFMULCSH __m128h _mm_maskz_mul_sch (__mmask8 k, __m128h a, __m128h b);
VFMULCSH __m128h _mm_mul_sch (__m128h a, __m128h b);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-60, “Type E10 Class Exception Conditions.”

Additionally:

#UD	If (dest_reg == src1_reg) or (dest_reg == src2_reg).

VFIXUPIMMPD—Fix Up Special Packed Float64 Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 54 /r ib VFIXUPIMMPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Fix up special numbers in float64 vector xmm1, float64 vector xmm2 and int64 vector xmm3/m128/m64bcst and store the result in xmm1, under writemask.
EVEX.256.66.0F3A.W1 54 /r ib VFIXUPIMMPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Fix up special numbers in float64 vector ymm1, float64 vector ymm2 and int64 vector ymm3/m256/m64bcst and store the result in ymm1, under writemask.
EVEX.512.66.0F3A.W1 54 /r ib VFIXUPIMMPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}, imm8	A	V/V	AVX512F OR AVX10.1	Fix up elements of float64 vector in zmm2 using int64 vector table in zmm3/m512/m64bcst, combine with preserved elements from zmm1, and store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Perform fix-up of quad-word elements encoded in double precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the corresponding quadword element of the second source operand (the third operand) with exception reporting specifier imm8. The elements that are fixed-up are selected by mask bits of 1 specified in the opmask k1. Mask bits of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up elements from the first source operand and the preserved element in the first operand are combined as the final results in the destination operand (the first operand).

The destination and the first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The two-level look-up table perform a fix-up of each double precision floating-point input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMPD can be used after the N-R reciprocal sequence to set the result to the correct value (i.e., INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below).

MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in the destination with the corresponding bit clear in k1 retain their previous values or are set to 0.

Operation

```
enum TOKEN_TYPE
```

```
{
    QNAN_TOKEN := 0,
    SNAN_TOKEN := 1,
    ZERO_VALUE_TOKEN := 2,
    POS_ONE_VALUE_TOKEN := 3,
    NEG_INF_TOKEN := 4,
    POS_INF_TOKEN := 5,
    NEG_VALUE_TOKEN := 6,
    POS_VALUE_TOKEN := 7
}

FIXUPIMM_DP (dest[63:0], src1[63:0],tbl3[63:0], imm8 [7:0]){
    tsrc[63:0] := ((src1[62:52] = 0) AND (MXCSR.DAZ =1)) ? 0.0 : src1[63:0]
    CASE(tsrc[63:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j := 0;
        SNAN_TOKEN: j := 1;
        ZERO_VALUE_TOKEN: j := 2;
        POS_ONE_VALUE_TOKEN: j := 3;
        NEG_INF_TOKEN: j := 4;
        POS_INF_TOKEN: j := 5;
        NEG_VALUE_TOKEN: j := 6;
        POS_VALUE_TOKEN: j := 7;
    }    ; end source special CASE(tsrc...)

    ; The required response from src3 table is extracted
    token_response[3:0] = tbl3[3+4*j:4*j];

    CASE(token_response[3:0]) {
        0000: dest[63:0] := dest[63:0];          ; preserve content of DEST
        0001: dest[63:0] := tsrc[63:0];          ; pass through src1 normal input value, denormal as zero
        0010: dest[63:0] := QNaN(tsrc[63:0]);
        0011: dest[63:0] := QNAN_Indefinite;
        0100: dest[63:0] := -INF;
        0101: dest[63:0] := +INF;
        0110: dest[63:0] := tsrc.sign? –INF : +INF;
        0111: dest[63:0] := -0;
        1000: dest[63:0] := +0;
        1001: dest[63:0] := -1;
        1010: dest[63:0] := +1;
        1011: dest[63:0] := ½;
        1100: dest[63:0] := 90.0;
        1101: dest[63:0] := PI/2;
        1110: dest[63:0] := MAX_FLOAT;
        1111: dest[63:0] := -MAX_FLOAT;
    }          ; end of token_response CASE

    ; The required fault reporting from imm8 is extracted
    ; TOKENs are mutually exclusive and TOKENs priority defines the order.
    ; Multiple faults related to a single token can occur simultaneously.
    IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
    IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
    IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
```

```
    IF (tsrc[63:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
        ; end fault reporting
    return dest[63:0];
}       ; end of FIXUPIMM_DP()
```

VFIXUPIMMPD

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := FIXUPIMM_DP(DEST[i+63:i], SRC1[i+63:i], SRC2[63:0], imm8 [7:0])
                ELSE
                    DEST[i+63:i] := FIXUPIMM_DP(DEST[i+63:i], SRC1[i+63:i], SRC2[i+63:i], imm8 [7:0])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] := 0             ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Immediate Control Description:

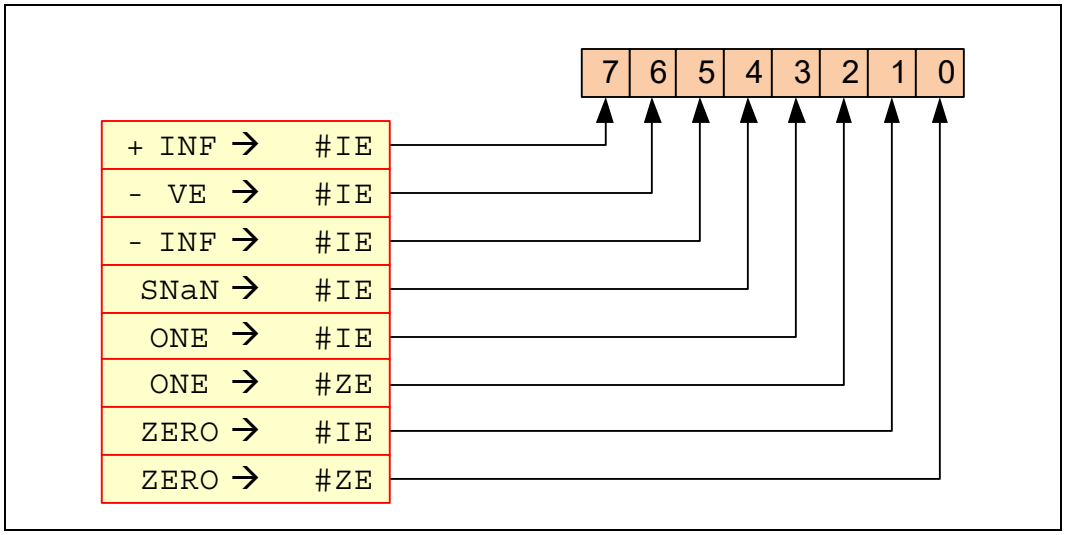


Figure 5-9. VFIXUPIMMPD Immediate Control Description

Intel C/C++ Compiler Intrinsic Equivalent

VFIXUPIMMPD __m512d _mm512_fixupimm_pd(__m512d a, __m512d b, __m512i c, int imm8);
VFIXUPIMMPD __m512d _mm512_mask_fixupimm_pd(__m512d a, __mmask8 k, __m512d b, __m512i c, int imm8);

VFIXUPIMMPD __m512d _mm512_maskz_fixupimm_pd(__mmask8 k, __m512d a, __m512d b, __m512i c, int imm8);
VFIXUPIMMPD __m512d _mm512_fixupimm_round_pd(__m512d a, __m512d b, __m512i c, int imm8, int sae);
VFIXUPIMMPD __m512d _mm512_mask_fixupimm_round_pd(__m512d a, __mmask8 k, __m512d b, __m512i c, int imm8, int sae);
VFIXUPIMMPD __m512d _mm512_maskz_fixupimm_round_pd(__mmask8 k, __m512d a, __m512d b, __m512i c, int imm8, int sae);
VFIXUPIMMPD __m256d _mm256_fixupimm_pd(__m256d a, m256d b, __m256i c, int imm8);
VFIXUPIMMPD __m256d _mm256_mask_fixupimm_pd(__m256d a, __mmask8 k, __m256d b, __m256i c, int imm8);
VFIXUPIMMPD __m256d _mm256_maskz_fixupimm_pd(__mmask8 k, __m256d a, __m256d b, __m256i c, int imm8);
VFIXUPIMMPD __m128d _mm_fixupimm_pd(__m128d a, __m128d b, __m128i c, int imm8);
VFIXUPIMMPD __m128d _mm_mask_fixupimm_pd(__m128d a, __mmask8 k, __m128d b, __m128i c, int imm8);
VFIXUPIMMPD __m128d _mm_maskz_fixupimm_pd(__mmask8 k, __m128d a, __m128d b, __m128i c, int imm8);

SIMD Floating-Point Exceptions

Zero, Invalid.

Other Exceptions

See Table 2-48, "Type E2 Class Exception Conditions."

VFIXUPIMMPS—Fix Up Special Packed Float32 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 54 /r VFIXUPIMMPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Fix up special numbers in float32 vector xmm1, float32 vector xmm2 and int32 vector xmm3/m128/m32bcst and store the result in xmm1, under writemask.
EVEX.256.66.0F3A.W0 54 /r VFIXUPIMMPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Fix up special numbers in float32 vector ymm1, float32 vector ymm2 and int32 vector ymm3/m256/m32bcst and store the result in ymm1, under writemask.
EVEX.512.66.0F3A.W0 54 /r ib VFIXUPIMMPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}, imm8	A	V/V	AVX512F OR AVX10.1	Fix up elements of float32 vector in zmm2 using int32 vector table in zmm3/m512/m32bcst, combine with preserved elements from zmm1, and store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Perform fix-up of doubleword elements encoded in single precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the corresponding doubleword element of the second source operand (the third operand) with exception reporting specifier imm8. The elements that are fixed-up are selected by mask bits of 1 specified in the opmask k1. Mask bits of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up elements from the first source operand and the preserved element in the first operand are combined as the final results in the destination operand (the first operand).

The destination and the first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The two-level look-up table perform a fix-up of each single precision floating-point input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMPS can be used after the N-R reciprocal sequence to set the result to the correct value (i.e., INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below).

MXCSR.DAZ is used and refer to zmm2 only (i.e., zmm1 is not considered as zero in case MXCSR.DAZ is set).

MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

Operation

```
enum TOKEN_TYPE
{
    QNAN_TOKEN := 0,
    SNAN_TOKEN := 1,
    ZERO_VALUE_TOKEN := 2,
    POS_ONE_VALUE_TOKEN := 3,
    NEG_INF_TOKEN := 4,
    POS_INF_TOKEN := 5,
    NEG_VALUE_TOKEN := 6,
    POS_VALUE_TOKEN := 7
}

FIXUPIMM_SP ( dest[31:0], src1[31:0],tbl3[31:0], imm8 [7:0]){
    tsrc[31:0] := ((src1[30:23] = 0) AND (MXCSR.DAZ =1)) ? 0.0 : src1[31:0]
    CASE(tsrc[31:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j := 0;
        SNAN_TOKEN: j := 1;
        ZERO_VALUE_TOKEN: j := 2;
        POS_ONE_VALUE_TOKEN: j := 3;
        NEG_INF_TOKEN: j := 4;
        POS_INF_TOKEN: j := 5;
        NEG_VALUE_TOKEN: j := 6;
        POS_VALUE_TOKEN: j := 7;
    }           ; end source special CASE(tsrc...)

    ; The required response from src3 table is extracted
    token_response[3:0] = tbl3[3+4*j:4*j];

    CASE(token_response[3:0]) {
        0000: dest[31:0] := dest[31:0];         ; preserve content of DEST
        0001: dest[31:0] := tsrc[31:0];         ; pass through src1 normal input value, denormal as zero
        0010: dest[31:0] := QNaN(tsrc[31:0]);
        0011: dest[31:0] := QNAN_Indefinite;
        0100: dest[31:0] := -INF;
        0101: dest[31:0] := +INF;
        0110: dest[31:0] := tsrc.sign? –INF : +INF;
        0111: dest[31:0] := -0;
        1000: dest[31:0] := +0;
        1001: dest[31:0] := -1;
        1010: dest[31:0] := +1;
        1011:  dest[31:0] := ½;
        1100: dest[31:0] := 90.0;
        1101: dest[31:0] := PI/2;
        1110: dest[31:0] := MAX_FLOAT;
        1111: dest[31:0] := -MAX_FLOAT;
    }           ; end of token_response CASE

    ; The required fault reporting from imm8 is extracted
    ; TOKENs are mutually exclusive and TOKENs priority defines the order.
    ; Multiple faults related to a single token can occur simultaneously.
    IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
    IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
```

```
    IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
        ; end fault reporting
    return dest[31:0];
}       ; end of FIXUPIMM_SP()
```

VFIXUPIMMPS (EVEX)

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] := FIXUPIMM_SP(DEST[i+31:i], SRC1[i+31:i], SRC2[31:0], imm8 [7:0])
                ELSE
                    DEST[i+31:i] := FIXUPIMM_SP(DEST[i+31:i], SRC1[i+31:i], SRC2[i+31:i], imm8 [7:0])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] := 0         ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Immediate Control Description:

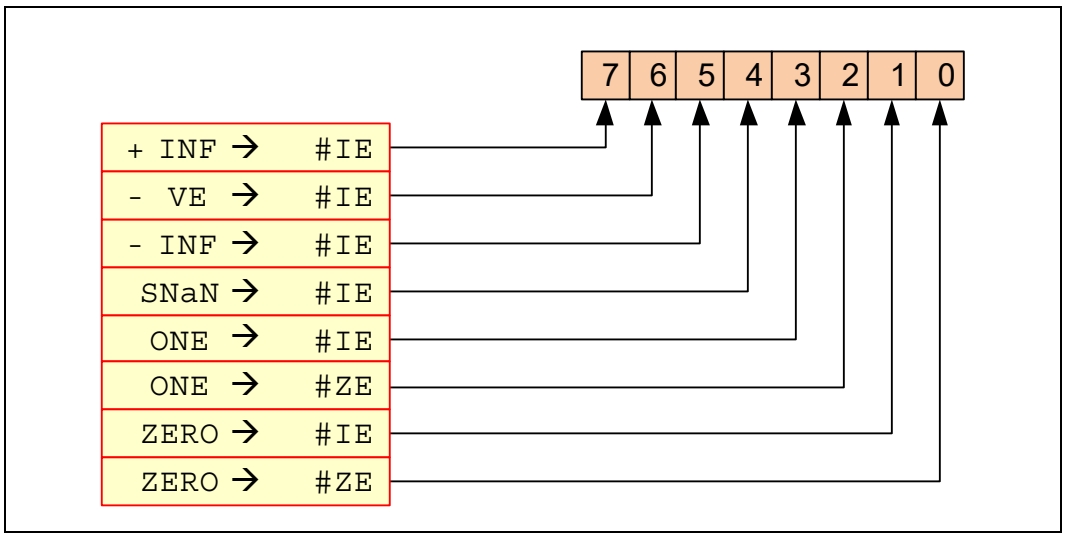


Figure 5-10. VFIXUPIMMPS Immediate Control Description

Intel C/C++ Compiler Intrinsic Equivalent

VFIXUPIMMPS __m512 _mm512_fixupimm_ps(__m512 a, __m512 b, __m512i c, int imm8);
VFIXUPIMMPS __m512 _mm512_mask_fixupimm_ps(__m512 a, __mmask16 k, __m512 b, __m512i c, int imm8);
VFIXUPIMMPS __m512 _mm512_maskz_fixupimm_ps(__mmask16 k, __m512 a, __m512 b, __m512i c, int imm8);
VFIXUPIMMPS __m512 _mm512_fixupimm_round_ps(__m512 a, __m512 b, __m512i c, int imm8, int sae);
VFIXUPIMMPS __m512 _mm512_mask_fixupimm_round_ps(__m512 a, __mmask16 k, __m512 b, __m512i c, int imm8, int sae);
VFIXUPIMMPS __m512 _mm512_maskz_fixupimm_round_ps(__mmask16 k, __m512 a, __m512 b, __m512i c, int imm8, int sae);
VFIXUPIMMPS __m256 _mm256_fixupimm_ps(__m256 a, __m256 b, __m256i c, int imm8);
VFIXUPIMMPS __m256 _mm256_mask_fixupimm_ps(__m256 a, __mmask8 k, __m256 b, __m256i c, int imm8);
VFIXUPIMMPS __m256 _mm256_maskz_fixupimm_ps(__mmask8 k, __m256 a, __m256 b, __m256i c, int imm8);
VFIXUPIMMPS __m128 _mm_fixupimm_ps(__m128 a, __m128 b, __m128i c, int imm8);
VFIXUPIMMPS __m128 _mm_mask_fixupimm_ps(__m128 a, __mmask8 k, __m128 b, __m128i c, int imm8);
VFIXUPIMMPS __m128 _mm_maskz_fixupimm_ps(__mmask8 k, __m128 a, __m128 b, __m128i c, int imm8);

SIMD Floating-Point Exceptions

Zero, Invalid.

Other Exceptions

See Table 2-48, "Type E2 Class Exception Conditions."

VFIXUPIMMSD—Fix Up Special Scalar Float64 Value

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W1 55 /r ib VFIXUPIMMSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	A	V/V	AVX512F OR AVX10.1	Fix up a float64 number in the low quadword element of xmm2 using scalar int32 table in xmm3/m64 and store the result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Perform a fix-up of the low quadword element encoded in double precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the low quadword element of the second source operand (the third operand) with exception reporting specifier imm8. The element that is fixed-up is selected by mask bit of 1 specified in the opmask k1. Mask bit of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up element from the first source operand or the preserved element in the first operand becomes the low quadword element of the destination operand (the first operand). Bits 127:64 of the destination operand is copied from the corresponding bits of the first source operand. The destination and first source operands are XMM registers. The second source operand can be a XMM register or a 64- bit memory location.

The two-level look-up table perform a fix-up of each double precision floating-point input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMPD can be used after the N-R reciprocal sequence to set the result to the correct value (i.e., INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below).

MXCSR.DAZ is used and refer to zmm2 only (i.e., zmm1 is not considered as zero in case MXCSR.DAZ is set).

MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

Operation

```
enum TOKEN_TYPE
{
    QNAN_TOKEN := 0,
    SNAN_TOKEN := 1,
    ZERO_VALUE_TOKEN := 2,
    POS_ONE_VALUE_TOKEN := 3,
    NEG_INF_TOKEN := 4,
    POS_INF_TOKEN := 5,
    NEG_VALUE_TOKEN := 6,
    POS_VALUE_TOKEN := 7
}

FIXUPIMM_DP (dest[63:0], src1[63:0],tbl3[63:0], imm8 [7:0]){
    tsrc[63:0] := ((src1[62:52] = 0) AND (MXCSR.DAZ =1)) ? 0.0 : src1[63:0]
    CASE(tsrc[63:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j := 0;
        SNAN_TOKEN: j := 1;
        ZERO_VALUE_TOKEN: j := 2;
        POS_ONE_VALUE_TOKEN: j := 3;
        NEG_INF_TOKEN: j := 4;
        POS_INF_TOKEN: j := 5;
        NEG_VALUE_TOKEN: j := 6;
        POS_VALUE_TOKEN: j := 7;
    }           ; end source special CASE(tsrc...)

    ; The required response from src3 table is extracted
    token_response[3:0] = tbl3[3+4*j:4*j];

    CASE(token_response[3:0]) {
        0000: dest[63:0] := dest[63:0]          ; preserve content of DEST
        0001: dest[63:0] := tsrc[63:0];         ; pass through src1 normal input value, denormal as zero
        0010: dest[63:0] := QNaN(tsrc[63:0]);
        0011: dest[63:0] := QNAN_Indefinite;
        0100:dest[63:0] := -INF;
        0101: dest[63:0] := +INF;
        0110: dest[63:0] := tsrc.sign? –INF : +INF;
        0111: dest[63:0] := -0;
        1000: dest[63:0] := +0;
        1001: dest[63:0] := -1;
        1010: dest[63:0] := +1;
        1011: dest[63:0] := ½;
        1100: dest[63:0] := 90.0;
        1101: dest[63:0] := PI/2;
        1110: dest[63:0] := MAX_FLOAT;
        1111: dest[63:0] := -MAX_FLOAT;
    }           ; end of token_response CASE

    ; The required fault reporting from imm8 is extracted
    ; TOKENs are mutually exclusive and TOKENs priority defines the order.
    ; Multiple faults related to a single token can occur simultaneously.
    IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
    IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
```

```
    IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
        ; end fault reporting
    return dest[63:0];
}       ; end of FIXUPIMM_DP()
```

VFIXUPIMMSD (EVEX encoded version)

```
IF k1[0] OR *no writemask*
    THEN DEST[63:0] := FIXUPIMM_DP(DEST[63:0], SRC1[63:0], SRC2[63:0], imm8 [7:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE  DEST[63:0] := 0               ; zeroing-masking
        FI
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Immediate Control Description:

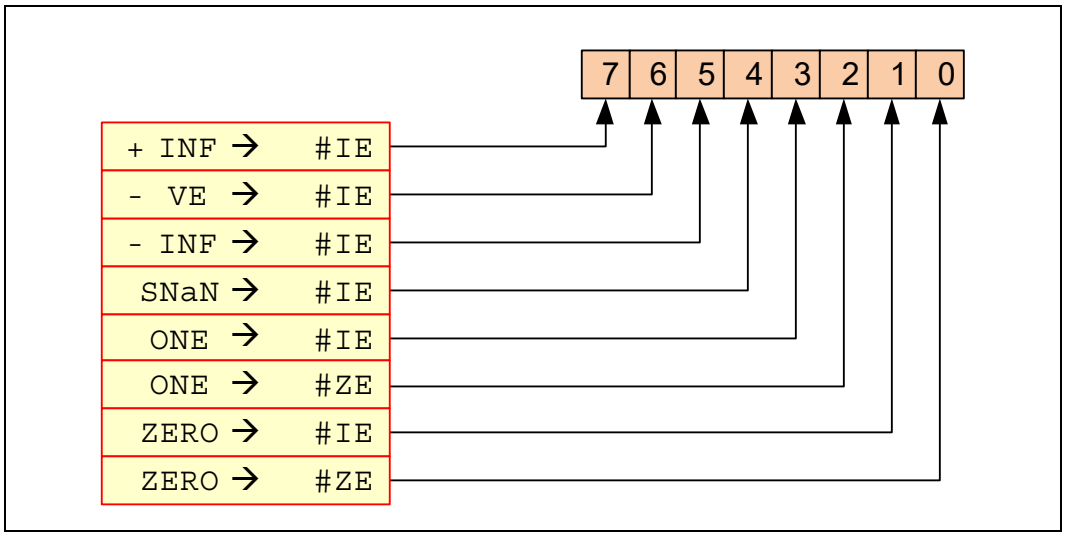


Figure 5-11. VFIXUPIMMSD Immediate Control Description

Intel C/C++ Compiler Intrinsic Equivalent

VFIXUPIMMSD __m128d _mm_fixupimm_sd(__m128d a, __m128d b, __m128i c, int imm8);
VFIXUPIMMSD __m128d _mm_mask_fixupimm_sd(__m128d a, __mmask8 k, __m128d b, __m128i c, int imm8);
VFIXUPIMMSD __m128d _mm_maskz_fixupimm_sd(__mmask8 k, __m128d a, __m128d b, __m128i c, int imm8);
VFIXUPIMMSD __m128d _mm_fixupimm_round_sd(__m128d a, __m128d b, __m128i c, int imm8, int sae);
VFIXUPIMMSD __m128d _mm_mask_fixupimm_round_sd(__m128d a, __mmask8 k, __m128d b, __m128i c, int imm8, int sae);
VFIXUPIMMSD __m128d _mm_maskz_fixupimm_round_sd(__mmask8 k, __m128d a, __m128d b, __m128i c, int imm8, int sae);

SIMD Floating-Point Exceptions

Zero, Invalid

Other Exceptions

See Table 2-49, "Type E3 Class Exception Conditions."

VFIXUPIMMSS—Fix Up Special Scalar Float32 Value

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W0 55 /r ib VFIXUPIMMSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512F OR AVX10.1	Fix up a float32 number in the low doubleword element in xmm2 using scalar int32 table in xmm3/m32 and store the result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Perform a fix-up of the low doubleword element encoded in single precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the low doubleword element of the second source operand (the third operand) with exception reporting specifier imm8. The element that is fixed-up is selected by mask bit of 1 specified in the opmask k1. Mask bit of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up element from the first source operand or the preserved element in the first operand becomes the low doubleword element of the destination operand (the first operand) Bits 127:32 of the destination operand is copied from the corresponding bits of the first source operand. The destination and first source operands are XMM registers. The second source operand can be a XMM register or a 32-bit memory location.

The two-level look-up table perform a fix-up of each single precision floating-point input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMPD can be used after the N-R reciprocal sequence to set the result to the correct value (i.e., INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below).

MXCSR.DAZ is used and refer to zmm2 only (i.e., zmm1 is not considered as zero in case MXCSR.DAZ is set).

MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

Operation

```
enum TOKEN_TYPE
{
    QNAN_TOKEN := 0,
    SNAN_TOKEN := 1,
    ZERO_VALUE_TOKEN := 2,
    POS_ONE_VALUE_TOKEN := 3,
    NEG_INF_TOKEN := 4,
    POS_INF_TOKEN := 5,
    NEG_VALUE_TOKEN := 6,
    POS_VALUE_TOKEN := 7
}

FIXUPIMM_SP (dest[31:0], src1[31:0],tbl3[31:0], imm8 [7:0]){
    tsrc[31:0] := ((src1[30:23] = 0) AND (MXCSR.DAZ =1)) ? 0.0 : src1[31:0]
    CASE(tsrc[63:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j := 0;
        SNAN_TOKEN: j := 1;
        ZERO_VALUE_TOKEN: j := 2;
        POS_ONE_VALUE_TOKEN: j := 3;
        NEG_INF_TOKEN: j := 4;
        POS_INF_TOKEN: j := 5;
        NEG_VALUE_TOKEN: j := 6;
        POS_VALUE_TOKEN: j := 7;
    }           ; end source special CASE(tsrc...)

    ; The required response from src3 table is extracted
    token_response[3:0] = tbl3[3+4*j:4*j];

    CASE(token_response[3:0]) {
        0000: dest[31:0] := dest[31:0];         ; preserve content of DEST
        0001: dest[31:0] := tsrc[31:0];         ; pass through src1 normal input value, denormal as zero
        0010: dest[31:0] := QNaN(tsrc[31:0]);
        0011: dest[31:0] := QNAN_Indefinite;
        0100: dest[31:0] := -INF;
        0101: dest[31:0] := +INF;
        0110: dest[31:0] := tsrc.sign? –INF : +INF;
        0111: dest[31:0] := -0;
        1000: dest[31:0] := +0;
        1001: dest[31:0] := -1;
        1010: dest[31:0] := +1;
        1011: dest[31:0] := ½;
        1100: dest[31:0] := 90.0;
        1101: dest[31:0] := PI/2;
        1110: dest[31:0] := MAX_FLOAT;
        1111: dest[31:0] := -MAX_FLOAT;
    }           ; end of token_response CASE

    ; The required fault reporting from imm8 is extracted
    ; TOKENs are mutually exclusive and TOKENs priority defines the order.
    ; Multiple faults related to a single token can occur simultaneously.
    IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
    IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
```

```
    IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
        ; end fault reporting
    return dest[31:0];
}       ; end of FIXUPIMM_SP()
```

VFIXUPIMMSS (EVEX encoded version)

```
IF k1[0] OR *no writemask*
    THEN DEST[31:0] := FIXUPIMM_SP(DEST[31:0], SRC1[31:0], SRC2[31:0], imm8 [7:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE  DEST[31:0] := 0               ; zeroing-masking
        FI
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Immediate Control Description:

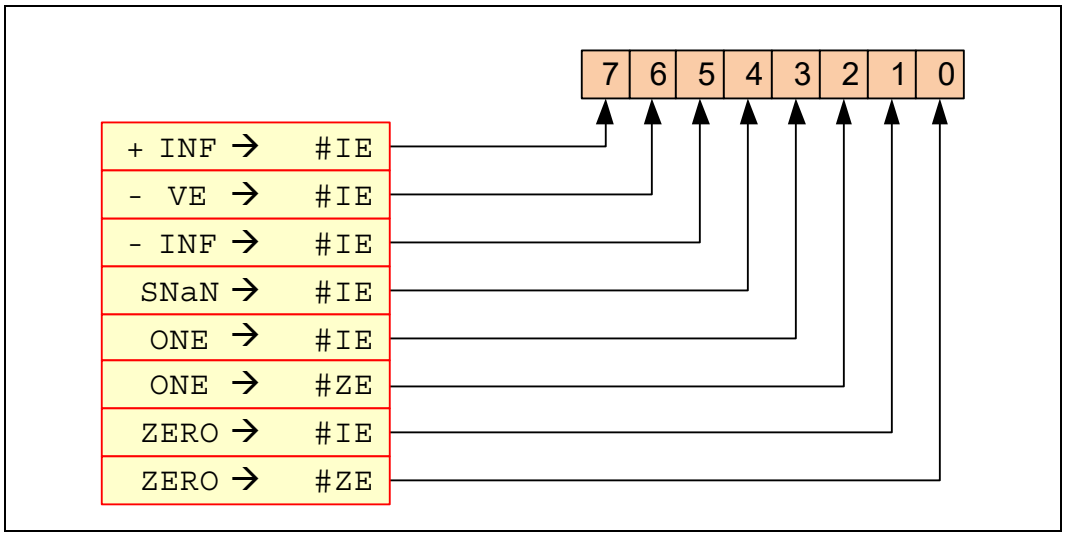


Figure 5-12. VFIXUPIMMSS Immediate Control Description

Intel C/C++ Compiler Intrinsic Equivalent

VFIXUPIMMSS __m128 _mm_fixupimm_ss(__m128 a, __m128 b, __m128i c, int imm8);
VFIXUPIMMSS __m128 _mm_mask_fixupimm_ss(__m128 a, __mmask8 k, __m128 b, __m128i c, int imm8);
VFIXUPIMMSS __m128 _mm_maskz_fixupimm_ss(__mmask8 k, __m128 a, __m128 b, __m128i c, int imm8);
VFIXUPIMMSS __m128 _mm_fixupimm_round_ss(__m128 a, __m128 b, __m128i c, int imm8, int sae);
VFIXUPIMMSS __m128 _mm_mask_fixupimm_round_ss(__m128 a, __mmask8 k, __m128 b, __m128i c, int imm8, int sae);
VFIXUPIMMSS __m128 _mm_maskz_fixupimm_round_ss(__mmask8 k, __m128 a, __m128 b, __m128i c, int imm8, int sae);

SIMD Floating-Point Exceptions

Zero, Invalid

Other Exceptions

See Table 2-49, "Type E3 Class Exception Conditions."

VFMADD132PD/VFMADD213PD/VFMADD231PD—Fused Multiply-Add of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 98 /r VFMADD132PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm1 and xmm3/mem, add to xmm2 and put result in xmm1.
VEX.128.66.0F38.W1 A8 /r VFMADD213PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm1 and xmm2, add to xmm3/mem and put result in xmm1.
VEX.128.66.0F38.W1 B8 /r VFMADD231PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm2 and xmm3/mem, add to xmm1 and put result in xmm1.
VEX.256.66.0F38.W1 98 /r VFMADD132PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm1 and ymm3/mem, add to ymm2 and put result in ymm1.
VEX.256.66.0F38.W1 A8 /r VFMADD213PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm1 and ymm2, add to ymm3/mem and put result in ymm1.
VEX.256.66.0F38.W1 B8 /r VFMADD231PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm2 and ymm3/mem, add to ymm1 and put result in ymm1.
EVEX.128.66.0F38.W1 98 /r VFMADD132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm1 and xmm3/m128/m64bcst, add to xmm2 and put result in xmm1.
EVEX.128.66.0F38.W1 A8 /r VFMADD213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm1 and xmm2, add to xmm3/m128/m64bcst and put result in xmm1.
EVEX.128.66.0F38.W1 B8 /r VFMADD231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm2 and xmm3/m128/m64bcst, add to xmm1 and put result in xmm1.
EVEX.256.66.0F38.W1 98 /r VFMADD132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm1 and ymm3/m256/m64bcst, add to ymm2 and put result in ymm1.
EVEX.256.66.0F38.W1 A8 /r VFMADD213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm1 and ymm2, add to ymm3/m256/m64bcst and put result in ymm1.
EVEX.256.66.0F38.W1 B8 /r VFMADD231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm2 and ymm3/m256/m64bcst, add to ymm1 and put result in ymm1.
EVEX.512.66.0F38.W1 98 /r VFMADD132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm1 and zmm3/m512/m64bcst, add to zmm2 and put result in zmm1.
EVEX.512.66.0F38.W1 A8 /r VFMADD213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm1 and zmm2, add to zmm3/m512/m64bcst and put result in zmm1.
EVEX.512.66.0F38.W1 B8 /r VFMADD231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm2 and zmm3/m512/m64bcst, add to zmm1 and put result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a set of SIMD multiply-add computation on packed double precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132PD: Multiplies the two, four or eight packed double precision floating-point values from the first source operand to the two, four or eight packed double precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two, four or eight packed double precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFMADD213PD: Multiplies the two, four or eight packed double precision floating-point values from the second source operand to the two, four or eight packed double precision floating-point values in the first source operand, adds the infinite precision intermediate result to the two, four or eight packed double precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFMADD231PD: Multiplies the two, four or eight packed double precision floating-point values from the second source to the two, four or eight packed double precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two, four or eight packed double precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) is a ZMM register and encoded in reg_field. The second source operand is a ZMM register and encoded in EVEX.vvvv. The third source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMADD132PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(DEST[n+63:n]*SRC3[n+63:n] + SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMADD213PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(SRC2[n+63:n]*DEST[n+63:n] + SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMADD231PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(SRC2[n+63:n]*SRC3[n+63:n] + DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[63:0])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] + DEST[i+63:i])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDxxxPD __m512d _mm512_fmadd_pd(__m512d a, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_fmadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_mask_fmadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_maskz_fmadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_mask3_fmadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMADDxxxPD __m512d _mm512_mask_fmadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_maskz_fmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_mask3_fmadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMADDxxxPD __m256d _mm256_mask_fmadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMADDxxxPD __m256d _mm256_maskz_fmadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMADDxxxPD __m256d _mm256_mask3_fmadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMADDxxxPD __m128d _mm_mask_fmadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDxxxPD __m128d _mm_maskz_fmadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDxxxPD __m128d _mm_mask3_fmadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDxxxPD __m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);
VFMADDxxxPD __m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, "Type 2 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

VF[,N]MADD[132,213,231]PH—Fused Multiply-Add of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP6.W0 98 /r VFMADD132PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, add to xmm2, and store the result in xmm1.
EVEX.256.66.MAP6.W0 98 /r VFMADD132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, add to ymm2, and store the result in ymm1.
EVEX.512.66.MAP6.W0 98 /r VFMADD132PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, add to zmm2, and store the result in zmm1.
EVEX.128.66.MAP6.W0 A8 /r VFMADD213PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm1 and xmm2, add to xmm3/m128/m16bcst, and store the result in xmm1.
EVEX.256.66.MAP6.W0 A8 /r VFMADD213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm1 and ymm2, add to ymm3/m256/m16bcst, and store the result in ymm1.
EVEX.512.66.MAP6.W0 A8 /r VFMADD213PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply packed FP16 values from zmm1 and zmm2, add to zmm3/m512/m16bcst, and store the result in zmm1.
EVEX.128.66.MAP6.W0 B8 /r VFMADD231PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm2 and xmm3/m128/m16bcst, add to xmm1, and store the result in xmm1.
EVEX.256.66.MAP6.W0 B8 /r VFMADD231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm2 and ymm3/m256/m16bcst, add to ymm1, and store the result in ymm1.
EVEX.512.66.MAP6.W0 B8 /r VFMADD231PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply packed FP16 values from zmm2 and zmm3/m512/m16bcst, add to zmm1, and store the result in zmm1.
EVEX.128.66.MAP6.W0 9C /r VFNMADD132PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, and negate the value. Add this value to xmm2, and store the result in xmm1.
EVEX.256.66.MAP6.W0 9C /r VFNMADD132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, and negate the value. Add this value to ymm2, and store the result in ymm1.
EVEX.512.66.MAP6.W0 9C /r VFNMADD132PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, and negate the value. Add this value to zmm2, and store the result in zmm1.
EVEX.128.66.MAP6.W0 AC /r VFNMADD213PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm1 and xmm2, and negate the value. Add this value to xmm3/m128/m16bcst, and store the result in xmm1.
EVEX.256.66.MAP6.W0 AC /r VFNMADD213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm1 and ymm2, and negate the value. Add this value to ymm3/m256/m16bcst, and store the result in ymm1.

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.MAP6.W0 AC /r VFNMADD213PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply packed FP16 values from zmm1 and zmm2, and negate the value. Add this value to zmm3/m512/m16bcst, and store the result in zmm1.
EVEX.128.66.MAP6.W0 BC /r VFNMADD231PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm2 and xmm3/m128/m16bcst, and negate the value. Add this value to xmm1, and store the result in xmm1.
EVEX.256.66.MAP6.W0 BC /r VFNMADD231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm2 and ymm3/m256/m16bcst, and negate the value. Add this value to ymm1, and store the result in ymm1.
EVEX.512.66.MAP6.W0 BC /r VFNMADD231PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply packed FP16 values from zmm2 and zmm3/m512/m16bcst, and negate the value. Add this value to zmm1, and store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a packed multiply-add or negated multiply-add computation on FP16 values using three source operands and writes the results in the destination operand. The destination operand is also the first source operand. The "N" (negated) forms of this instruction add the negated infinite precision intermediate product to the corresponding remaining operand. The notation' "132", "213" and "231" indicate the use of the operands in ±A * B + C, where each digit corresponds to the operand number, with the destination being operand 1; see Table 5-5.

The destination elements are updated according to the writemask.

Table 5-5. VF[,N]MADD[132,213,231]PH Notation for Operands

Notation	Operands
132	dest = ± dest*src3+src2
231	dest = ± src2*src3+dest
213	dest = ± src2*dest+src3

Operation

VF[,N]MADD132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-DEST.fp16[j]*SRC3.fp16[j] + SRC2.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j]*SRC3.fp16[j] + SRC2.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MADD132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-DEST.fp16[j] * t3 + SRC2.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * t3 + SRC2.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MADD213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j]*DEST.fp16[j] + SRC3.fp16[j])
        ELSE
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*DEST.fp16[j] + SRC3.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MADD213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j] * DEST.fp16[j] + t3 )
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * DEST.fp16[j] + t3 )
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MADD231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *negative form:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j]*SRC3.fp16[j] + DEST.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*SRC3.fp16[j] + DEST.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MADD231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j] * t3 + DEST.fp16[j] )
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * t3 + DEST.fp16[j] )
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADD132PH, VFMADD213PH , and VFMADD231PH:
__m128h _mm_fmadd_ph (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fmadd_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fmadd_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fmadd_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h _mm256_fmadd_ph (__m256h a, __m256h b, __m256h c);
__m256h _mm256_mask_fmadd_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h _mm256_mask3_fmadd_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h _mm256_maskz_fmadd_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h _mm512_fmadd_ph (__m512h a, __m512h b, __m512h c);
__m512h _mm512_mask_fmadd_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h _mm512_mask3_fmadd_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h _mm512_maskz_fmadd_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h _mm512_fmadd_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask_fmadd_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask3_fmadd_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h _mm512_maskz_fmadd_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);

VFNMADD132PH, VFNMADD213PH, and VFNMADD231PH:
__m128h _mm_fnmadd_ph (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fnmadd_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fnmadd_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fnmadd_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h _mm256_fnmadd_ph (__m256h a, __m256h b, __m256h c);
__m256h _mm256_mask_fnmadd_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h _mm256_mask3_fnmadd_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h _mm256_maskz_fnmadd_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h _mm512_fnmadd_ph (__m512h a, __m512h b, __m512h c);
__m512h _mm512_mask_fnmadd_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h _mm512_mask3_fnmadd_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h _mm512_maskz_fnmadd_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h _mm512_fnmadd_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask_fnmadd_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask3_fnmadd_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h _mm512_maskz_fnmadd_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VFMADD132PS/VFMADD213PS/VFMADD231PS—Fused Multiply-Add of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 98 /r VFMADD132PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm1 and xmm3/mem, add to xmm2 and put result in xmm1.
VEX.128.66.0F38.W0 A8 /r VFMADD213PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm1 and xmm2, add to xmm3/mem and put result in xmm1.
VEX.128.66.0F38.W0 B8 /r VFMADD231PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm2 and xmm3/mem, add to xmm1 and put result in xmm1.
VEX.256.66.0F38.W0 98 /r VFMADD132PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm1 and ymm3/mem, add to ymm2 and put result in ymm1.
VEX.256.66.0F38.W0 A8 /r VFMADD213PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm1 and ymm2, add to ymm3/mem and put result in ymm1.
VEX.256.66.0F38.0 B8 /r VFMADD231PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm2 and ymm3/mem, add to ymm1 and put result in ymm1.
EVEX.128.66.0F38.W0 98 /r VFMADD132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm1 and xmm3/m128/m32bcst, add to xmm2 and put result in xmm1.
EVEX.128.66.0F38.W0 A8 /r VFMADD213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm1 and xmm2, add to xmm3/m128/m32bcst and put result in xmm1.
EVEX.128.66.0F38.W0 B8 /r VFMADD231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm2 and xmm3/m128/m32bcst, add to xmm1 and put result in xmm1.
EVEX.256.66.0F38.W0 98 /r VFMADD132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm1 and ymm3/m256/m32bcst, add to ymm2 and put result in ymm1.
EVEX.256.66.0F38.W0 A8 /r VFMADD213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm1 and ymm2, add to ymm3/m256/m32bcst and put result in ymm1.
EVEX.256.66.0F38.W0 B8 /r VFMADD231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm2 and ymm3/m256/m32bcst, add to ymm1 and put result in ymm1.
EVEX.512.66.0F38.W0 98 /r VFMADD132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm1 and zmm3/m512/m32bcst, add to zmm2 and put result in zmm1.
EVEX.512.66.0F38.W0 A8 /r VFMADD213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm1 and zmm2, add to zmm3/m512/m32bcst and put result in zmm1.
EVEX.512.66.0F38.W0 B8 /r VFMADD231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm2 and zmm3/m512/m32bcst, add to zmm1 and put result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a set of SIMD multiply-add computation on packed single precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the first source operand to the four, eight or sixteen packed single precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFMADD213PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the four, eight or sixteen packed single precision floating-point values in the first source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single precision floating-point values in the third source operand, performs rounding and stores the resulting the four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFMADD231PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the four, eight or sixteen packed single precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) is a ZMM register and encoded in reg_field. The second source operand is a ZMM register and encoded in EVEX.vvvv. The third source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMADD132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM := 4
ELSEIF (VEX.256)
    MAXNUM := 8
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
```

```
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMADD213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM := 4
ELSEIF (VEX.256)
    MAXNUM := 8
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] + SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMADD231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM := 4
ELSEIF (VEX.256)
    MAXNUM := 8
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
```

```
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[i+31:i])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[31:0])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
```

```
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDxxxPS __m512 _mm512_fmadd_ps(__m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 _mm512_fmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 _mm512_mask_fmadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDxxxPS __m512 _mm512_maskz_fmadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 _mm512_mask3_fmadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDxxxPS __m512 _mm512_mask_fmadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 _mm512_maskz_fmadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 _mm512_mask3_fmadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMADDxxxPS __m256 _mm256_mask_fmadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMADDxxxPS __m256 _mm256_maskz_fmadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDxxxPS __m256 _mm256_mask3_fmadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDxxxPS __m128 _mm_mask_fmadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxxPS __m128 _mm_maskz_fmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m128 _mm_mask3_fmadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDxxxPS __m128 _mm_fmadd_ps (__m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VFMADD132SD/VFMADD213SD/VFMADD231SD—Fused Multiply-Add of Scalar Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W1 99 /r VFMADD132SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double precision floating-point value from xmm1 and xmm3/m64, add to xmm2 and put result in xmm1.
VEX.LIG.66.0F38.W1 A9 /r VFMADD213SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double precision floating-point value from xmm1 and xmm2, add to xmm3/m64 and put result in xmm1.
VEX.LIG.66.0F38.W1 B9 /r VFMADD231SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double precision floating-point value from xmm2 and xmm3/m64, add to xmm1 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 99 /r VFMADD132SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar double precision floating-point value from xmm1 and xmm3/m64, add to xmm2 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 A9 /r VFMADD213SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar double precision floating-point value from xmm1 and xmm2, add to xmm3/m64 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 B9 /r VFMADD231SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar double precision floating-point value from xmm2 and xmm3/m64, add to xmm1 and put result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD multiply-add computation on the low double precision floating-point values using three source operands and writes the multiply-add result in the destination operand. The destination operand is also the first source operand. The first and second operand are XMM registers. The third source operand can be an XMM register or a 64-bit memory location.

VFMADD132SD: Multiplies the low double precision floating-point value from the first source operand to the low double precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low double precision floating-point values in the second source operand, performs rounding and stores the resulting double precision floating-point value to the destination operand (first source operand).

VFMADD213SD: Multiplies the low double precision floating-point value from the second source operand to the low double precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low double precision floating-point value in the third source operand, performs rounding and stores the resulting double precision floating-point value to the destination operand (first source operand).

VFMADD231SD: Multiplies the low double precision floating-point value from the second source to the low double precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low double precision floating-point value in the first source operand, performs rounding and stores the resulting double precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination is updated according to the writemask.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMADD132SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFMADD213SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFMADD231SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFMADD132SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := MAXVL-1:128RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
DEST[127:63] := DEST[127:63]
DEST[MAXVL-1:128] := 0
```

VFMADD213SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
DEST[127:63] := DEST[127:63]
DEST[MAXVL-1:128] := 0
```

VFMADD231SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
DEST[127:63] := DEST[127:63]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFMADDxxxSD __m128d _mm_fmadd_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d _mm_mask_fmadd_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDxxxSD __m128d _mm_maskz_fmadd_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDxxxSD __m128d _mm_mask3_fmadd_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDxxxSD __m128d _mm_mask_fmadd_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d _mm_maskz_fmadd_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d _mm_mask3_fmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMADDxxxSD __m128d _mm_fmadd_sd (__m128d a, __m128d b, __m128d c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VF[,N]MADD[132,213,231]SH—Fused Multiply-Add of Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.MAP6.W0 99 /r VFMADD132SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply FP16 values from xmm1 and xmm3/m16, add to xmm2, and store the result in xmm1.
EVEX.LLIG.66.MAP6.W0 A9 /r VFMADD213SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply FP16 values from xmm1 and xmm2, add to xmm3/m16, and store the result in xmm1.
EVEX.LLIG.66.MAP6.W0 B9 /r VFMADD231SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply FP16 values from xmm2 and xmm3/m16, add to xmm1, and store the result in xmm1.
EVEX.LLIG.66.MAP6.W0 9D /r VFNMADD132SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply FP16 values from xmm1 and xmm3/m16, and negate the value. Add this value to xmm2, and store the result in xmm1.
EVEX.LLIG.66.MAP6.W0 AD /r VFNMADD213SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply FP16 values from xmm1 and xmm2, and negate the value. Add this value to xmm3/m16, and store the result in xmm1.
EVEX.LLIG.66.MAP6.W0 BD /r VFNMADD231SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply FP16 values from xmm2 and xmm3/m16, and negate the value. Add this value to xmm1, and store the result in xmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a scalar multiply-add or negated multiply-add computation on the low FP16 values using three source operands and writes the result in the destination operand. The destination operand is also the first source operand. The "N" (negated) forms of this instruction add the negated infinite precision intermediate product to the corresponding remaining operand. The notation' "132", "213" and "231" indicate the use of the operands in ±A * B + C, where each digit corresponds to the operand number, with the destination being operand 1; see Table 5-6.

Bits 127:16 of the destination operand are preserved. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Table 5-6. VF[,N]MADD[132,213,231]SH Notation for Operands

Notation	Operands
132	dest = ± dest*src3+src2
231	dest = ± src2*src3+dest
213	dest = ± src2*dest+src3

Operation

VF[,N]MADD132SH DEST, SRC2, SRC3 (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC3 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    IF *negative form*:
        DEST.fp16[0] := RoundFPControl(-DEST.fp16[0]*SRC3.fp16[0] + SRC2.fp16[0])
    ELSE:
        DEST.fp16[0] := RoundFPControl(DEST.fp16[0]*SRC3.fp16[0] + SRC2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged
DEST[MAXVL-1:128] := 0
```

VF[,N]MADD213SH DEST, SRC2, SRC3 (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC3 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    IF *negative form:
        DEST.fp16[0] := RoundFPControl(-SRC2.fp16[0]*DEST.fp16[0] + SRC3.fp16[0])
    ELSE:
        DEST.fp16[0] := RoundFPControl(SRC2.fp16[0]*DEST.fp16[0] + SRC3.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged
DEST[MAXVL-1:128] := 0
```

VF[,N]MADD231SH DEST, SRC2, SRC3 (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC3 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    IF *negative form*:
        DEST.fp16[0] := RoundFPControl(-SRC2.fp16[0]*SRC3.fp16[0] + DEST.fp16[0])
    ELSE:
        DEST.fp16[0] := RoundFPControl(SRC2.fp16[0]*SRC3.fp16[0] + DEST.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADD132SH, VFMADD213SH, and VFMADD231SH:
__m128h _mm_fmadd_round_sh (__m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask_fmadd_round_sh (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask3_fmadd_round_sh (__m128h a, __m128h b, __m128h c, __mmask8 k, const int rounding);
__m128h _mm_maskz_fmadd_round_sh (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_fmadd_sh (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fmadd_sh (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fmadd_sh (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fmadd_sh (__mmask8 k, __m128h a, __m128h b, __m128h c);

VFNMADD132SH, VFNMADD213SH, and VFNMADD231SH:
__m128h _mm_fnmadd_round_sh (__m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask_fnmadd_round_sh (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask3_fnmadd_round_sh (__m128h a, __m128h b, __m128h c, __mmask8 k, const int rounding);
__m128h _mm_maskz_fnmadd_round_sh (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_fnmadd_sh (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fnmadd_sh (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fnmadd_sh (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fnmadd_sh (__mmask8 k, __m128h a, __m128h b, __m128h c);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VFMADD132SS/VFMADD213SS/VFMADD231SS—Fused Multiply-Add of Scalar Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W0 99 /r VFMADD132SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single precision floating-point value from xmm1 and xmm3/m32, add to xmm2 and put result in xmm1.
VEX.LIG.66.0F38.W0 A9 /r VFMADD213SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single precision floating-point value from xmm1 and xmm2, add to xmm3/m32 and put result in xmm1.
VEX.LIG.66.0F38.W0 B9 /r VFMADD231SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single precision floating-point value from xmm2 and xmm3/m32, add to xmm1 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 99 /r VFMADD132SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar single precision floating-point value from xmm1 and xmm3/m32, add to xmm2 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 A9 /r VFMADD213SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar single precision floating-point value from xmm1 and xmm2, add to xmm3/m32 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 B9 /r VFMADD231SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar single precision floating-point value from xmm2 and xmm3/m32, add to xmm1 and put result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD multiply-add computation on single precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The first and second operands are XMM registers. The third source operand can be a XMM register or a 32-bit memory location.

VFMADD132SS: Multiplies the low single precision floating-point value from the first source operand to the low single precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low single precision floating-point value in the second source operand, performs rounding and stores the resulting single precision floating-point value to the destination operand (first source operand).

VFMADD213SS: Multiplies the low single precision floating-point value from the second source operand to the low single precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low single precision floating-point value in the third source operand, performs rounding and stores the resulting single precision floating-point value to the destination operand (first source operand).

VFMADD231SS: Multiplies the low single precision floating-point value from the second source operand to the low single precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low single precision floating-point value in the first source operand, performs rounding and stores the resulting single precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMADD132SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(DEST[31:0]*SRC3[31:0] + SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMADD213SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(SRC2[31:0]*DEST[31:0] + SRC3[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMADD231SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(SRC2[31:0]*SRC3[31:0] + DEST[31:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0]] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMADD132SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(DEST[31:0]*SRC3[31:0] + SRC2[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMADD213SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] + SRC3[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMADD231SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(SRC2[31:0]*SRC3[31:0] + DEST[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFMADDxxxSS __m128 _mm_fmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 _mm_mask_fmadd_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxxSS __m128 _mm_maskz_fmadd_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDxxxSS __m128 _mm_mask3_fmadd_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDxxxSS __m128 _mm_mask_fmadd_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 _mm_maskz_fmadd_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 _mm_mask3_fmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFMADDxxxSS __m128 _mm_fmadd_ss (__m128 a, __m128 b, __m128 c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD—Fused Multiply-Alternating Add/Subtract of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 96 /r VFMADDSUB132PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm1 and xmm3/mem, add/subtract elements in xmm2 and put result in xmm1.
VEX.128.66.0F38.W1 A6 /r VFMADDSUB213PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/mem and put result in xmm1.
VEX.128.66.0F38.W1 B6 /r VFMADDSUB231PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm2 and xmm3/mem, add/subtract elements in xmm1 and put result in xmm1.
VEX.256.66.0F38.W1 96 /r VFMADDSUB132PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm1 and ymm3/mem, add/subtract elements in ymm2 and put result in ymm1.
VEX.256.66.0F38.W1 A6 /r VFMADDSUB213PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/mem and put result in ymm1.
VEX.256.66.0F38.W1 B6 /r VFMADDSUB231PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm2 and ymm3/mem, add/subtract elements in ymm1 and put result in ymm1.
EVEX.128.66.0F38.W1 A6 /r VFMADDSUB213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.
EVEX.128.66.0F38.W1 B6 /r VFMADDSUB231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm2 and xmm3/m128/m64bcst, add/subtract elements in xmm1 and put result in xmm1 subject to writemask k1.
EVEX.128.66.0F38.W1 96 /r VFMADDSUB132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm1 and xmm3/m128/m64bcst, add/subtract elements in xmm2 and put result in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W1 A6 /r VFMADDSUB213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1.
EVEX.256.66.0F38.W1 B6 /r VFMADDSUB231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm2 and ymm3/m256/m64bcst, add/subtract elements in ymm1 and put result in ymm1 subject to writemask k1.
EVEX.256.66.0F38.W1 96 /r VFMADDSUB132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm1 and ymm3/m256/m64bcst, add/subtract elements in ymm2 and put result in ymm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 A6 /r VFMADDSUB213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm1and zmm2, add/subtract elements in zmm3/m512/m64bcst and put result in zmm1 subject to writemask k1.
EVEX.512.66.0F38.W1 B6 /r VFMADDSUB231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm2 and zmm3/m512/m64bcst, add/subtract elements in zmm1 and put result in zmm1 subject to writemask k1.
EVEX.512.66.0F38.W1 96 /r VFMADDSUB132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm1 and zmm3/m512/m64bcst, add/subtract elements in zmm2 and put result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFMADDSUB132PD: Multiplies the two, four, or eight packed double precision floating-point values from the first source operand to the two or four packed double precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double precision floating-point elements and subtracts the even double precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double precision floating-point values to the destination operand (first source operand).

VFMADDSUB213PD: Multiplies the two, four, or eight packed double precision floating-point values from the second source operand to the two or four packed double precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd double precision floating-point elements and subtracts the even double precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double precision floating-point values to the destination operand (first source operand).

VFMADDSUB231PD: Multiplies the two, four, or eight packed double precision floating-point values from the second source operand to the two or four packed double precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double precision floating-point elements and subtracts the even double precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFMADDSUB132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] := RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
    DEST[MAXVL-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[MAXVL-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[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
    DEST[191:128] := RoundFPControl_MXCSR(SRC2[191:128]*SRC3[191:128] - DEST[191:128])
    DEST[255:192] := RoundFPControl_MXCSR(SRC2[255:192]*SRC3[255:192] + DEST[255:192]
FI
```

VFMADDSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] :=
                    RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
                ELSE DEST[i+63:i] :=
                    RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] - SRC2[i+63:i])
                        ELSE
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
                FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
                        ELSE
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
                FI;
```

```
            FI

        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] :=
                    RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
                ELSE DEST[i+63:i] :=
                    RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0])
                        ELSE
```

```
                        DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[63:0])
                        ELSE
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
                    FI;
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] :=
                    RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
                ELSE DEST[i+63:i] :=
                    RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                        RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])
                        ELSE
                            DEST[i+63:i] :=
                        RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                        RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] + DEST[i+63:i])
                        ELSE
                            DEST[i+63:i] :=
                        RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
                    FI;
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDSUBxxxPD __m512d _mm512_fmaddsub_pd(__m512d a, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d _mm512_fmaddsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d _mm512_mask_fmaddsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d _mm512_maskz_fmaddsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d _mm512_mask3_fmaddsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMADDSUBxxxPD __m512d _mm512_mask_fmaddsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d _mm512_maskz_fmaddsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d _mm512_mask3_fmaddsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMADDSUBxxxPD __m256d _mm256_mask_fmaddsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMADDSUBxxxPD __m256d _mm256_maskz_fmaddsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMADDSUBxxxPD __m256d _mm256_mask3_fmaddsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMADDSUBxxxPD __m128d _mm_mask_fmaddsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDSUBxxxPD __m128d _mm_maskz_fmaddsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDSUBxxxPD __m128d _mm_mask3_fmaddsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDSUBxxxPD __m128d _mm_fmaddsub_pd (__m128d a, __m128d b, __m128d c);
VFMADDSUBxxxPD __m256d _mm256_fmaddsub_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, "Type 2 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

VFMADDSUB132PH/VFMADDSUB213PH/VFMADDSUB231PH—Fused Multiply-Alternating Add/Subtract of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP6.W0 96 /r VFMADDSUB132PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, add/subtract elements in xmm2, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 96 /r VFMADDSUB132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, add/subtract elements in ymm2, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 96 /r VFMADDSUB132PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, add/subtract elements in zmm2, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 A6 /r VFMADDSUB213PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm1 and xmm2, add/subtract elements in xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 A6 /r VFMADDSUB213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm1 and ymm2, add/subtract elements in ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 A6 /r VFMADDSUB213PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply packed FP16 values from zmm1 and zmm2, add/subtract elements in zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 B6 /r VFMADDSUB231PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm2 and xmm3/m128/m16bcst, add/subtract elements in xmm1, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 B6 /r VFMADDSUB231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm2 and ymm3/m256/m16bcst, add/subtract elements in ymm1, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 B6 /r VFMADDSUB231PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply packed FP16 values from zmm2 and zmm3/m512/m16bcst, add/subtract elements in zmm1, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a packed multiply-add (odd elements) or multiply-subtract (even elements) computation on FP16 values using three source operands and writes the results in the destination operand. The destination operand is also the first source operand. The notation' "132", "213" and "231" indicate the use of the operands in A * B ± C, where each digit corresponds to the operand number, with the destination being operand 1; see Table 5-10.

The destination elements are updated according to the writemask.

Table 5-7. VFMADDSUB[132,213,231]PH Notation for Odd and Even Elements

Notation	Odd Elements	Even Elements
132	dest = dest*src3+src2	dest = dest*src3-src2
231	dest = src2*src3+dest	dest = src2*src3-dest
213	dest = src2*dest+src3	dest = src2*dest-src3

Operation

VFMADDSUB132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * SRC3.fp16[j] - SRC2.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * SRC3.fp16[j] + SRC2.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VFMADDSUB132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * t3 - SRC2.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * t3 + SRC2.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
```

```
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VFMADDSUB213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*DEST.fp16[j] - SRC3.fp16[j])
        ELSE
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*DEST.fp16[j] + SRC3.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VFMADDSUB213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * DEST.fp16[j] - t3)
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * DEST.fp16[j] + t3)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VFMADDSUB231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *j is even:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * SRC3.fp16[j] - DEST.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * SRC3.fp16[j] + DEST.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VFMADDSUB231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * t3 - DEST.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * t3 + DEST.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDSUB132PH, VFMADDSUB213PH, and VFMADDSUB231PH:
__m128h _mm_fmaddsub_ph (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fmaddsub_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fmaddsub_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fmaddsub_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h _mm256_fmaddsub_ph (__m256h a, __m256h b, __m256h c);
__m256h _mm256_mask_fmaddsub_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h _mm256_mask3_fmaddsub_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h _mm256_maskz_fmaddsub_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h _mm512_fmaddsub_ph (__m512h a, __m512h b, __m512h c);
__m512h _mm512_mask_fmaddsub_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h _mm512_mask3_fmaddsub_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h _mm512_maskz_fmaddsub_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h _mm512_fmaddsub_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask_fmaddsub_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask3_fmaddsub_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h _mm512_maskz_fmaddsub_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VFMADDSUB132PS/VFMADDSUB213PS/VFMADDSUB231PS—Fused Multiply-Alternating Add/Subtract of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 96 /r VFMADDSUB132PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm1 and xmm3/mem, add/subtract elements in xmm2 and put result in xmm1.
VEX.128.66.0F38.W0 A6 /r VFMADDSUB213PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/mem and put result in xmm1.
VEX.128.66.0F38.W0 B6 /r VFMADDSUB231PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm2 and xmm3/mem, add/subtract elements in xmm1 and put result in xmm1.
VEX.256.66.0F38.W0 96 /r VFMADDSUB132PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm1 and ymm3/mem, add/subtract elements in ymm2 and put result in ymm1.
VEX.256.66.0F38.W0 A6 /r VFMADDSUB213PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/mem and put result in ymm1.
VEX.256.66.0F38.W0 B6 /r VFMADDSUB231PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm2 and ymm3/mem, add/subtract elements in ymm1 and put result in ymm1.
EVEX.128.66.0F38.W0 A6 /r VFMADDSUB213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/m128/m32bcst and put result in xmm1 subject to writemask k1.
EVEX.128.66.0F38.W0 B6 /r VFMADDSUB231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm2 and xmm3/m128/m32bcst, add/subtract elements in xmm1 and put result in xmm1 subject to writemask k1.
EVEX.128.66.0F38.W0 96 /r VFMADDSUB132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm1 and xmm3/m128/m32bcst, add/subtract elements in zmm2 and put result in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 A6 /r VFMADDSUB213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/m256/m32bcst and put result in ymm1 subject to writemask k1.
EVEX.256.66.0F38.W0 B6 /r VFMADDSUB231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm2 and ymm3/m256/m32bcst, add/subtract elements in ymm1 and put result in ymm1 subject to writemask k1.
EVEX.256.66.0F38.W0 96 /r VFMADDSUB132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm1 and ymm3/m256/m32bcst, add/subtract elements in ymm2 and put result in ymm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 A6 /r VFMADDSUB213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm1 and zmm2, add/subtract elements in zmm3/m512/m32bcst and put result in zmm1 subject to writemask k1.
EVEX.512.66.0F38.W0 B6 /r VFMADDSUB231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm2 and zmm3/m512/m32bcst, add/subtract elements in zmm1 and put result in zmm1 subject to writemask k1.
EVEX.512.66.0F38.W0 96 /r VFMADDSUB132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm1 and zmm3/m512/m32bcst, add/subtract elements in zmm2 and put result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFMADDSUB132PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the first source operand to the corresponding packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single precision floating-point elements and subtracts the even single precision floating-point values in the second source operand, performs rounding and stores the resulting packed single precision floating-point values to the destination operand (first source operand).

VFMADDSUB213PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the corresponding packed single precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd single precision floating-point elements and subtracts the even single precision floating-point values in the third source operand, performs rounding and stores the resulting packed single precision floating-point values to the destination operand (first source operand).

VFMADDSUB231PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the corresponding packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single precision floating-point elements and subtracts the even single precision floating-point values in the first source operand, performs rounding and stores the resulting packed single precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMADDSUB132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM :=2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM -1{
    n := 64*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
    DEST[n+63:n+32] := RoundFPControl_MXCSR(DEST[n+63:n+32]*SRC3[n+63:n+32] + SRC2[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMADDSUB213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM -1{
    n := 64*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
    DEST[n+63:n+32] := RoundFPControl_MXCSR(SRC2[n+63:n+32]*DEST[n+63:n+32] + SRC3[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMADDSUB231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM -1{
    n := 64*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
    DEST[n+63:n+32] :=RoundFPControl_MXCSR(SRC2[n+63:n+32]*SRC3[n+63:n+32] + DEST[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMADDSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) (4, 128), (8, 256),= (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] :=
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
                ELSE DEST[i+31:i] :=
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] - SRC2[i+31:i])
                        ELSE
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[i+31:i])
                        ELSE
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
```

```
                    FI;
            FI

        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
                ELSE DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[31:0])
```

```
                        ELSE
                            DEST[i+31:i] :=
                        RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
                        FI;
                    ELSE
                        IF (EVEX.b = 1)
                            THEN
                                DEST[i+31:i] :=
                        RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[31:0])
                            ELSE
                                DEST[i+31:i] :=
                        RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
                        FI;
                FI
            ELSE
                IF *merging-masking*                ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE                            ; zeroing-masking
                        DEST[i+31:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
                ELSE DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
```

```
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
                        ELSE
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
                        ELSE
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
                    FI;
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDSUBxxxPS __m512 _mm512_fmaddsub_ps(__m512 a, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 _mm512_fmaddsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512 _mm512_mask_fmaddsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 _mm512_maskz_fmaddsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 _mm512_mask3_fmaddsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDSUBxxxPS __m512 _mm512_mask_fmaddsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512 _mm512_maskz_fmaddsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512 _mm512_mask3_fmaddsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMADDSUBxxxPS __m256 _mm256_mask_fmaddsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMADDSUBxxxPS __m256 _mm256_maskz_fmaddsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDSUBxxxPS __m256 _mm256_mask3_fmaddsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDSUBxxxPS __m128 _mm_mask_fmaddsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDSUBxxxPS __m128 _mm_maskz_fmaddsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDSUBxxxPS __m128 _mm_mask3_fmaddsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDSUBxxxPS __m128 _mm_fmaddsub_ps (__m128 a, __m128 b, __m128 c);
VFMADDSUBxxxPS __m256 _mm256_fmaddsub_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, "Type 2 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

VFMSUB132PD/VFMSUB213PD/VFMSUB231PD—Fused Multiply-Subtract of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 9A /r VFMSUB132PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm1 and xmm3/mem, subtract xmm2 and put result in xmm1.
VEX.128.66.0F38.W1 AA /r VFMSUB213PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm1 and xmm2, subtract xmm3/mem and put result in xmm1.
VEX.128.66.0F38.W1 BA /r VFMSUB231PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm2 and xmm3/mem, subtract xmm1 and put result in xmm1.
VEX.256.66.0F38.W1 9A /r VFMSUB132PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm1 and ymm3/mem, subtract ymm2 and put result in ymm1.
VEX.256.66.0F38.W1 AA /r VFMSUB213PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm1 and ymm2, subtract ymm3/mem and put result in ymm1.
VEX.256.66.0F38.W1 BA /r VFMSUB231PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm2 and ymm3/mem, subtract ymm1 and put result in ymm1.S
EVEX.128.66.0F38.W1 9A /r VFMSUB132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm1 and xmm3/m128/m64bcst, subtract xmm2 and put result in xmm1 subject to writemask k1.
EVEX.128.66.0F38.W1 AA /r VFMSUB213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm1 and xmm2, subtract xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.
EVEX.128.66.0F38.W1 BA /r VFMSUB231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm2 and xmm3/m128/m64bcst, subtract xmm1 and put result in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W1 9A /r VFMSUB132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm1 and ymm3/m256/m64bcst, subtract ymm2 and put result in ymm1 subject to writemask k1.
EVEX.256.66.0F38.W1 AA /r VFMSUB213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm1 and ymm2, subtract ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1.
EVEX.256.66.0F38.W1 BA /r VFMSUB231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm2 and ymm3/m256/m64bcst, subtract ymm1 and put result in ymm1 subject to writemask k1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 9A /r VFMSUB132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm1 and zmm3/m512/m64bcst, subtract zmm2 and put result in zmm1 subject to writemask k1.
EVEX.512.66.0F38.W1 AA /r VFMSUB213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm1 and zmm2, subtract zmm3/m512/m64bcst and put result in zmm1 subject to writemask k1.
EVEX.512.66.0F38.W1 BA /r VFMSUB231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm2 and zmm3/m512/m64bcst, subtract zmm1 and put result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a set of SIMD multiply-subtract computation on packed double precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132PD: Multiplies the two, four or eight packed double precision floating-point values from the first source operand to the two, four or eight packed double precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFMSUB213PD: Multiplies the two, four or eight packed double precision floating-point values from the second source operand to the two, four or eight packed double precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFMSUB231PD: Multiplies the two, four or eight packed double precision floating-point values from the second source to the two, four or eight packed double precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFMSUB132PD DEST, SRC2, SRC3 (VEX Encoded Versions)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(DEST[n+63:n]*SRC3[n+63:n] - SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUB213PD DEST, SRC2, SRC3 (VEX Encoded Versions)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(SRC2[n+63:n]*DEST[n+63:n] - SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUB231PD DEST, SRC2, SRC3 (VEX Encoded Versions)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(SRC2[n+63:n]*SRC3[n+63:n] - DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUB132PD DEST, SRC2, SRC3 (EVEX Encoded Versions, When SRC3 Operand is a Register)

(KL, VL) = (2, 128), (4, 256), (8, 512)

```
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB132PD DEST, SRC2, SRC3 (EVEX Encoded Versions, When SRC3 Operand is a Memory Source)

(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] - SRC2[i+63:i])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
            FI;
            ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB213PD DEST, SRC2, SRC3 (EVEX Encoded Versions, When SRC3 Operand is a Register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB213PD DEST, SRC2, SRC3 (EVEX Encoded Versions, When SRC3 Operand is a Memory Source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0])
+31:i])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB231PD DEST, SRC2, SRC3 (EVEX Encoded Versions, When SRC3 Operand is a Register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB231PD DEST, SRC2, SRC3 (EVEX Encoded Versions, When SRC3 Operand is a Memory Source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBxxxPD __m512d _mm512_fmsub_pd(__m512d a, __m512d b, __m512d c);
VFMSUBxxxPD __m512d _mm512_fmsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d _mm512_mask_fmsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMSUBxxxPD __m512d _mm512_maskz_fmsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMSUBxxxPD __m512d _mm512_mask3_fmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMSUBxxxPD __m512d _mm512_mask_fmsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d _mm512_maskz_fmsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d _mm512_mask3_fmsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMSUBxxxPD __m256d _mm256_mask_fmsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMSUBxxxPD __m256d _mm256_maskz_fmsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMSUBxxxPD __m256d _mm256_mask3_fmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMSUBxxxPD __m128d _mm_mask_fmsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBxxxPD __m128d _mm_maskz_fmsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBxxxPD __m128d _mm_mask3_fmsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxPD __m128d _mm_fmsub_pd (__m128d a, __m128d b, __m128d c);
VFMSUBxxxPD __m256d _mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VF[,N]MSUB[132,213,231]PH—Fused Multiply-Subtract of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP6.W0 9A /r VFMSUB132PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, subtract xmm2, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 9A /r VFMSUB132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, subtract ymm2, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 9A /r VFMSUB132PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, subtract zmm2, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 AA /r VFMSUB213PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm1 and xmm2, subtract xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 AA /r VFMSUB213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm1 and ymm2, subtract ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 AA /r VFMSUB213PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply packed FP16 values from zmm1 and zmm2, subtract zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 BA /r VFMSUB231PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm2 and xmm3/m128/m16bcst, subtract xmm1, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 BA /r VFMSUB231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm2 and ymm3/m256/m16bcst, subtract ymm1, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 BA /r VFMSUB231PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply packed FP16 values from zmm2 and zmm3/m512/m16bcst, subtract zmm1, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 9E /r VFNMSUB132PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, and negate the value. Subtract xmm2 from this value, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 9E /r VFNMSUB132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, and negate the value. Subtract ymm2 from this value, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 9E /r VFNMSUB132PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, and negate the value. Subtract zmm2 from this value, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 AE /r VFNMSUB213PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm1 and xmm2, and negate the value. Subtract xmm3/m128/m16bcst from this value, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 AE /r VFNMSUB213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm1 and ymm2, and negate the value. Subtract ymm3/m256/m16bcst from this value, and store the result in ymm1 subject to writemask k1.

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.MAP6.W0 AE /r VFNMSUB213PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply packed FP16 values from zmm1 and zmm2, and negate the value. Subtract zmm3/m512/m16bcst from this value, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 BE /r VFNMSUB231PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm2 and xmm3/m128/m16bcst, and negate the value. Subtract xmm1 from this value, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 BE /r VFNMSUB231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm2 and ymm3/m256/m16bcst, and negate the value. Subtract ymm1 from this value, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 BE /r VFNMSUB231PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply packed FP16 values from zmm2 and zmm3/m512/m16bcst, and negate the value. Subtract zmm1 from this value, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a packed multiply-subtract or a negated multiply-subtract computation on FP16 values using three source operands and writes the results in the destination operand. The destination operand is also the first source operand. The "N" (negated) forms of this instruction subtract the remaining operand from the negated infinite precision intermediate product. The notation' "132", "213" and "231" indicate the use of the operands in ±A * B − C, where each digit corresponds to the operand number, with the destination being operand 1; see Table 5-8.

The destination elements are updated according to the writemask.

Table 5-8. VF[,N]MSUB[132,213,231]PH Notation for Operands

Notation	Operands
132	dest = ± dest*src3-src2
231	dest = ± src2*src3-dest
213	dest = ± src2*dest-src3

Operation

VF[,N]MSUB132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-DEST.fp16[j]*SRC3.fp16[j] - SRC2.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j]*SRC3.fp16[j] - SRC2.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MSUB132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-DEST.fp16[j] * t3 - SRC2.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * t3 - SRC2.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MSUB213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j]*DEST.fp16[j] - SRC3.fp16[j])
        ELSE
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*DEST.fp16[j] - SRC3.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MSUB213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j] * DEST.fp16[j] - t3 )
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * DEST.fp16[j] - t3 )
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MSUB231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *negative form:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j]*SRC3.fp16[j] - DEST.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*SRC3.fp16[j] - DEST.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MSUB231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j] * t3 - DEST.fp16[j] )
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * t3 - DEST.fp16[j] )
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUB132PH, VFMSUB213PH, and VFMSUB231PH:
__m128h _mm_fmsub_ph (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fmsub_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fmsub_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fmsub_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h _mm256_fmsub_ph (__m256h a, __m256h b, __m256h c);
__m256h _mm256_mask_fmsub_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h _mm256_mask3_fmsub_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h _mm256_maskz_fmsub_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h _mm512_fmsub_ph (__m512h a, __m512h b, __m512h c);
__m512h _mm512_mask_fmsub_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h _mm512_mask3_fmsub_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h _mm512_maskz_fmsub_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h _mm512_fmsub_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask_fmsub_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask3_fmsub_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h _mm512_maskz_fmsub_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);

VFNMSUB132PH, VFNMSUB213PH, and VFNMSUB231PH:
__m128h _mm_fnmsub_ph (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fnmsub_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fnmsub_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fnmsub_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h _mm256_fnmsub_ph (__m256h a, __m256h b, __m256h c);
__m256h _mm256_mask_fnmsub_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h _mm256_mask3_fnmsub_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h _mm256_maskz_fnmsub_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h _mm512_fnmsub_ph (__m512h a, __m512h b, __m512h c);
__m512h _mm512_mask_fnmsub_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h _mm512_mask3_fnmsub_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h _mm512_maskz_fnmsub_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h _mm512_fnmsub_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask_fnmsub_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask3_fnmsub_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h _mm512_maskz_fnmsub_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

VFMSUB132PS/VFMSUB213PS/VFMSUB231PS—Fused Multiply-Subtract of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 9A /r VFMSUB132PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm1 and xmm3/mem, subtract xmm2 and put result in xmm1.
VEX.128.66.0F38.W0 AA /r VFMSUB213PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm1 and xmm2, subtract xmm3/mem and put result in xmm1.
VEX.128.66.0F38.W0 BA /r VFMSUB231PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm2 and xmm3/mem, subtract xmm1 and put result in xmm1.
VEX.256.66.0F38.W0 9A /r VFMSUB132PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm1 and ymm3/mem, subtract ymm2 and put result in ymm1.
VEX.256.66.0F38.W0 AA /r VFMSUB213PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm1 and ymm2, subtract ymm3/mem and put result in ymm1.
VEX.256.66.0F38.0 BA /r VFMSUB231PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm2 and ymm3/mem, subtract ymm1 and put result in ymm1.
EVEX.128.66.0F38.W0 9A /r VFMSUB132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm1 and xmm3/m128/m32bcst, subtract xmm2 and put result in xmm1.
EVEX.128.66.0F38.W0 AA /r VFMSUB213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm1 and xmm2, subtract xmm3/m128/m32bcst and put result in xmm1.
EVEX.128.66.0F38.W0 BA /r VFMSUB231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm2 and xmm3/m128/m32bcst, subtract xmm1 and put result in xmm1.
EVEX.256.66.0F38.W0 9A /r VFMSUB132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm1 and ymm3/m256/m32bcst, subtract ymm2 and put result in ymm1.
EVEX.256.66.0F38.W0 AA /r VFMSUB213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm1 and ymm2, subtract ymm3/m256/m32bcst and put result in ymm1.
EVEX.256.66.0F38.W0 BA /r VFMSUB231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm2 and ymm3/m256/m32bcst, subtract ymm1 and put result in ymm1.
EVEX.512.66.0F38.W0 9A /r VFMSUB132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm1 and zmm3/m512/m32bcst, subtract zmm2 and put result in zmm1.
EVEX.512.66.0F38.W0 AA /r VFMSUB213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm1 and zmm2, subtract zmm3/m512/m32bcst and put result in zmm1.
EVEX.512.66.0F38.W0 BA /r VFMSUB231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm2 and zmm3/m512/m32bcst, subtract zmm1 and put result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a set of SIMD multiply-subtract computation on packed single precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the first source operand to the four, eight or sixteen packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFMSUB213PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the four, eight or sixteen packed single precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFMSUB231PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source to the four, eight or sixteen packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFMSUB132PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUB213PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUB231PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] - SRC2[i+31:i])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[31:0])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBxxxPS __m512 _mm512_fmsub_ps(__m512 a, __m512 b, __m512 c);
VFMSUBxxxPS __m512 _mm512_fmsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 _mm512_mask_fmsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMSUBxxxPS __m512 _mm512_maskz_fmsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMSUBxxxPS __m512 _mm512_mask3_fmsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMSUBxxxPS __m512 _mm512_mask_fmsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 _mm512_maskz_fmsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 _mm512_mask3_fmsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMSUBxxxPS __m256 _mm256_mask_fmsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMSUBxxxPS __m256 _mm256_maskz_fmsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMSUBxxxPS __m256 _mm256_mask3_fmsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMSUBxxxPS __m128 _mm_mask_fmsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBxxxPS __m128 _mm_maskz_fmsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBxxxPS __m128 _mm_mask3_fmsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBxxxPS __m128 _mm_fmsub_ps (__m128 a, __m128 b, __m128 c);
VFMSUBxxxPS __m256 _mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VFMSUB132SD/VFMSUB213SD/VFMSUB231SD—Fused Multiply-Subtract of Scalar Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W1 9B /r VFMSUB132SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double precision floating-point value from xmm1 and xmm3/m64, subtract xmm2 and put result in xmm1.
VEX.LIG.66.0F38.W1 AB /r VFMSUB213SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double precision floating-point value from xmm1 and xmm2, subtract xmm3/m64 and put result in xmm1.
VEX.LIG.66.0F38.W1 BB /r VFMSUB231SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double precision floating-point value from xmm2 and xmm3/m64, subtract xmm1 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 9B /r VFMSUB132SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar double precision floating-point value from xmm1 and xmm3/m64, subtract xmm2 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 AB /r VFMSUB213SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar double precision floating-point value from xmm1 and xmm2, subtract xmm3/m64 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 BB /r VFMSUB231SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar double precision floating-point value from xmm2 and xmm3/m64, subtract xmm1 and put result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD multiply-subtract computation on the low packed double precision floating-point values using three source operands and writes the multiply-subtract result in the destination operand. The destination operand is also the first source operand. The second operand must be a XMM register. The third source operand can be a XMM register or a 64-bit memory location.

VFMSUB132SD: Multiplies the low packed double precision floating-point value from the first source operand to the low packed double precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double precision floating-point values in the second source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VFMSUB213SD: Multiplies the low packed double precision floating-point value from the second source operand to the low packed double precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts the low packed double precision floating-point value in the third source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VFMSUB231SD: Multiplies the low packed double precision floating-point value from the second source to the low packed double precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double precision floating-point value in the first source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFMSUB132SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFMSUB213SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFMSUB132SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFMSUB213SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFMSUB231SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBxxxSD __m128d _mm_fmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d _mm_mask_fmsub_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBxxxSD __m128d _mm_maskz_fmsub_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBxxxSD __m128d _mm_mask3_fmsub_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxSD __m128d _mm_mask_fmsub_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d _mm_maskz_fmsub_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d _mm_mask3_fmsub_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMSUBxxxSD __m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VF[,N]MSUB[132,213,231]SH—Fused Multiply-Subtract of Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.MAP6.W0 9B /r VFMSUB132SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply FP16 values from xmm1 and xmm3/m16, subtract xmm2, and store the result in xmm1 subject to writemask k1.
EVEX.LLIG.66.MAP6.W0 AB /r VFMSUB213SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply FP16 values from xmm1 and xmm2, subtract xmm3/m16, and store the result in xmm1 subject to writemask k1.
EVEX.LLIG.66.MAP6.W0 BB /r VFMSUB231SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply FP16 values from xmm2 and xmm3/m16, subtract xmm1, and store the result in xmm1 subject to writemask k1.
EVEX.LLIG.66.MAP6.W0 9F /r VFNMSUB132SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply FP16 values from xmm1 and xmm3/m16, and negate the value. Subtract xmm2 from this value, and store the result in xmm1 subject to writemask k1.
EVEX.LLIG.66.MAP6.W0 AF /r VFNMSUB213SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply FP16 values from xmm1 and xmm2, and negate the value. Subtract xmm3/m16 from this value, and store the result in xmm1 subject to writemask k1.
EVEX.LLIG.66.MAP6.W0 BF /r VFNMSUB231SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply FP16 values from xmm2 and xmm3/m16, and negate the value. Subtract xmm1 from this value, and store the result in xmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a scalar multiply-subtract or negated multiply-subtract computation on the low FP16 values using three source operands and writes the result in the destination operand. The destination operand is also the first source operand. The "N" (negated) forms of this instruction subtract the remaining operand from the negated infinite precision intermediate product. The notation' "132", "213" and "231" indicate the use of the operands in ±A * B − C, where each digit corresponds to the operand number, with the destination being operand 1; see Table 5-9.

Bits 127:16 of the destination operand are preserved. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Table 5-9. VF[,N]MSUB[132,213,231]SH Notation for Operands

Notation	Operands
132	dest = ± dest*src3-src2
231	dest = ± src2*src3-dest
213	dest = ± src2*dest-src3

Operation

VF[,N]MSUB132SH DEST, SRC2, SRC3 (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC3 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    IF *negative form*:
        DEST.fp16[0] := RoundFPControl(-DEST.fp16[0]*SRC3.fp16[0] - SRC2.fp16[0])
    ELSE:
        DEST.fp16[0] := RoundFPControl(DEST.fp16[0]*SRC3.fp16[0] - SRC2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged
DEST[MAXVL-1:128] := 0
```

VF[,N]MSUB213SH DEST, SRC2, SRC3 (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC3 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    IF *negative form:
        DEST.fp16[0] := RoundFPControl(-SRC2.fp16[0]*DEST.fp16[0] - SRC3.fp16[0])
    ELSE:
        DEST.fp16[0] := RoundFPControl(SRC2.fp16[0]*DEST.fp16[0] - SRC3.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged
DEST[MAXVL-1:128] := 0
```

VF[,N]MSUB231SH DEST, SRC2, SRC3 (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC3 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    IF *negative form*:
        DEST.fp16[0] := RoundFPControl(-SRC2.fp16[0]*SRC3.fp16[0] - DEST.fp16[0])
    ELSE:
        DEST.fp16[0] := RoundFPControl(SRC2.fp16[0]*SRC3.fp16[0] - DEST.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUB132SH, VFMSUB213SH, and VFMSUB231SH:
__m128h _mm_fmsub_round_sh (__m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask_fmsub_round_sh (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask3_fmsub_round_sh (__m128h a, __m128h b, __m128h c, __mmask8 k, const int rounding);
__m128h _mm_maskz_fmsub_round_sh (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_fmsub_sh (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fmsub_sh (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fmsub_sh (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fmsub_sh (__mmask8 k, __m128h a, __m128h b, __m128h c);

VFNMSUB132SH, VFNMSUB213SH, and VFNMSUB231SH:
__m128h _mm_fnmsub_round_sh (__m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask_fnmsub_round_sh (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask3_fnmsub_round_sh (__m128h a, __m128h b, __m128h c, __mmask8 k, const int rounding);
__m128h _mm_maskz_fnmsub_round_sh (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_fnmsub_sh (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fnmsub_sh (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fnmsub_sh (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fnmsub_sh (__mmask8 k, __m128h a, __m128h b, __m128h c);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VFMSUB132SS/VFMSUB213SS/VFMSUB231SS—Fused Multiply-Subtract of Scalar Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W0 9B /r VFMSUB132SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single precision floating-point value from xmm1 and xmm3/m32, subtract xmm2 and put result in xmm1.
VEX.LIG.66.0F38.W0 AB /r VFMSUB213SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single precision floating-point value from xmm1 and xmm2, subtract xmm3/m32 and put result in xmm1.
VEX.LIG.66.0F38.W0 BB /r VFMSUB231SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single precision floating-point value from xmm2 and xmm3/m32, subtract xmm1 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 9B /r VFMSUB132SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar single precision floating-point value from xmm1 and xmm3/m32, subtract xmm2 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 AB /r VFMSUB213SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar single precision floating-point value from xmm1 and xmm2, subtract xmm3/m32 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 BB /r VFMSUB231SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar single precision floating-point value from xmm2 and xmm3/m32, subtract xmm1 and put result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD multiply-subtract computation on the low packed single precision floating-point values using three source operands and writes the multiply-subtract result in the destination operand. The destination operand is also the first source operand. The second operand must be a XMM register. The third source operand can be a XMM register or a 32-bit memory location.

VFMSUB132SS: Multiplies the low packed single precision floating-point value from the first source operand to the low packed single precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single precision floating-point values in the second source operand, performs rounding and stores the resulting packed single precision floating-point value to the destination operand (first source operand).

VFMSUB213SS: Multiplies the low packed single precision floating-point value from the second source operand to the low packed single precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts the low packed single precision floating-point value in the third source operand, performs rounding and stores the resulting packed single precision floating-point value to the destination operand (first source operand).

VFMSUB231SS: Multiplies the low packed single precision floating-point value from the second source to the low packed single precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single precision floating-point value in the first source operand, performs rounding and stores the resulting packed single precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFMSUB132SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(DEST[31:0]*SRC3[31:0] - SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMSUB213SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(SRC2[31:0]*DEST[31:0] - SRC3[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMSUB231SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(SRC2[31:0]*SRC3[63:0] - DEST[31:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMSUB132SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(DEST[31:0]*SRC3[31:0] - SRC2[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMSUB213SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] - SRC3[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMSUB231SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(SRC2[31:0]*SRC3[31:0] - DEST[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBxxxSS __m128 _mm_fmsub_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_mask_fmsub_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBxxxSS __m128 _mm_maskz_fmsub_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBxxxSS __m128 _mm_mask3_fmsub_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBxxxSS __m128 _mm_mask_fmsub_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_maskz_fmsub_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_mask3_fmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFMSUBxxxSS __m128 _mm_fmsub_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD—Fused Multiply-Alternating Subtract/Add of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 97 /r VFMSUBADD132PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm1 and xmm3/mem, subtract/add elements in xmm2 and put result in xmm1.
VEX.128.66.0F38.W1 A7 /r VFMSUBADD213PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/mem and put result in xmm1.
VEX.128.66.0F38.W1 B7 /r VFMSUBADD231PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm2 and xmm3/mem, subtract/add elements in xmm1 and put result in xmm1.
VEX.256.66.0F38.W1 97 /r VFMSUBADD132PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm1 and ymm3/mem, subtract/add elements in ymm2 and put result in ymm1.
VEX.256.66.0F38.W1 A7 /r VFMSUBADD213PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/mem and put result in ymm1.
VEX.256.66.0F38.W1 B7 /r VFMSUBADD231PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm2 and ymm3/mem, subtract/add elements in ymm1 and put result in ymm1.
EVEX.128.66.0F38.W1 97 /r VFMSUBADD132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm1 and xmm3/m128/m64bcst, subtract/add elements in xmm2 and put result in xmm1 subject to writemask k1.
EVEX.128.66.0F38.W1 A7 /r VFMSUBADD213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.
EVEX.128.66.0F38.W1 B7 /r VFMSUBADD231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm2 and xmm3/m128/m64bcst, subtract/add elements in xmm1 and put result in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W1 97 /r VFMSUBADD132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm1 and ymm3/m256/m64bcst, subtract/add elements in ymm2 and put result in ymm1 subject to writemask k1.
EVEX.256.66.0F38.W1 A7 /r VFMSUBADD213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1.
EVEX.256.66.0F38.W1 B7 /r VFMSUBADD231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm2 and ymm3/m256/m64bcst, subtract/add elements in ymm1 and put result in ymm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 97 /r VFMSUBADD132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm1 and zmm3/m512/m64bcst, subtract/add elements in zmm2 and put result in zmm1 subject to writemask k1.
EVEX.512.66.0F38.W1 A7 /r VFMSUBADD213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm1 and zmm2, subtract/add elements in zmm3/m512/m64bcst and put result in zmm1 subject to writemask k1.
EVEX.512.66.0F38.W1 B7 /r VFMSUBADD231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm2 and zmm3/m512/m64bcst, subtract/add elements in zmm1 and put result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFMSUBADD132PD: Multiplies the two, four, or eight packed double precision floating-point values from the first source operand to the two or four packed double precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double precision floating-point elements and adds the even double precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double precision floating-point values to the destination operand (first source operand).

VFMSUBADD213PD: Multiplies the two, four, or eight packed double precision floating-point values from the second source operand to the two or four packed double precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd double precision floating-point elements and adds the even double precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double precision floating-point values to the destination operand (first source operand).

VFMSUBADD231PD: Multiplies the two, four, or eight packed double precision floating-point values from the second source operand to the two or four packed double precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double precision floating-point elements and adds the even double precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations

involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMSUBADD132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] := RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] - SRC2[127:64])
    DEST[MAXVL-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[MAXVL-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[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] - DEST[127:64])
    DEST[191:128] := RoundFPControl_MXCSR(SRC2[191:128]*SRC3[191:128] + DEST[191:128])
    DEST[255:192] := RoundFPControl_MXCSR(SRC2[255:192]*SRC3[255:192] - DEST[255:192]
FI
```

VFMSUBADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
```

```
        THEN
            IF j *is even*
                THEN DEST[i+63:i] :=
                    RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
                ELSE DEST[i+63:i] :=
                    RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
                        ELSE
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] - SRC2[i+63:i])
                        ELSE
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
                    FI;
            FI

        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] :=
                    RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
                ELSE DEST[i+63:i] :=
                    RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[63:0])
                        ELSE
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0])
                        ELSE
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
```

```
                    FI;
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] :=
                    RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
                ELSE DEST[i+63:i] :=
                    RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] + DEST[i+63:i])
                        ELSE
```

```
                        DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])

                        ELSE
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
                    FI;
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBADDxxxPD __m512d _mm512_fmsubadd_pd(__m512d a, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d _mm512_fmsubadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d _mm512_mask_fmsubadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d _mm512_maskz_fmsubadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d _mm512_mask3_fmsubadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMSUBADDxxxPD __m512d _mm512_mask_fmsubadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d _mm512_maskz_fmsubadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d _mm512_mask3_fmsubadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMSUBADDxxxPD __m256d _mm256_mask_fmsubadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMSUBADDxxxPD __m256d _mm256_maskz_fmsubadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMSUBADDxxxPD __m256d _mm256_mask3_fmsubadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMSUBADDxxxPD __m128d _mm_mask_fmsubadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBADDxxxPD __m128d _mm_maskz_fmsubadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBADDxxxPD __m128d _mm_mask3_fmsubadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBADDxxxPD __m128d _mm_fmsubadd_pd (__m128d a, __m128d b, __m128d c);
VFMSUBADDxxxPD __m256d _mm256_fmsubadd_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VFMSUBADD132PH/VFMSUBADD213PH/VFMSUBADD231PH—Fused Multiply-Alternating Subtract/Add of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP6.W0 97 /r VFMSUBADD132PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, subtract/add elements in xmm2, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 97 /r VFMSUBADD132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, subtract/add elements in ymm2, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 97 /r VFMSUBADD132PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, subtract/add elements in zmm2, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 A7 /r VFMSUBADD213PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm1 and xmm2, subtract/add elements in xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 A7 /r VFMSUBADD213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm1 and ymm2, subtract/add elements in ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 A7 /r VFMSUBADD213PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply packed FP16 values from zmm1 and zmm2, subtract/add elements in zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 B7 /r VFMSUBADD231PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm2 and xmm3/m128/m16bcst, subtract/add elements in xmm1, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 B7 /r VFMSUBADD231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm2 and ymm3/m256/m16bcst, subtract/add elements in ymm1, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 B7 /r VFMSUBADD231PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply packed FP16 values from zmm2 and zmm3/m512/m16bcst, subtract/add elements in zmm1, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a packed multiply-add (even elements) or multiply-subtract (odd elements) computation on FP16 values using three source operands and writes the results in the destination operand. The destination operand is also the first source operand. The notation "132", "213" and "231" indicate the use of the operands in A * B ± C, where each digit corresponds to the operand number, with the destination being operand 1; see Table 5-10.

The destination elements are updated according to the writemask.

Table 5-10. VFMSUBADD[132,213,231]PH Notation for Odd and Even Elements

Notation	Odd Elements	Even Elements
132	dest = dest*src3-src2	dest = dest*src3+src2
231	dest = src2*src3-dest	dest = src2*src3+dest
213	dest = src2*dest-src3	dest = src2*dest+src3

Operation

VFMSUBADD132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j]*SRC3.fp16[j] + SRC2.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j]*SRC3.fp16[j] - SRC2.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VFMSUBADD132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * t3 + SRC2.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * t3 - SRC2.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
```

```
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0:
```

VFMSUBADD213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*DEST.fp16[j] + SRC3.fp16[j])
        ELSE
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*DEST.fp16[j] - SRC3.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VFMSUBADD213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * DEST.fp16[j] + t3 )
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * DEST.fp16[j] - t3 )
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0:
```

VFMSUBADD231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *j is even:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*SRC3.fp16[j] + DEST.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*SRC3.fp16[j] - DEST.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VFMSUBADD231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * t3 + DEST.fp16[j] )
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * t3 - DEST.fp16[j] )
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBADD132PH, VFMSUBADD213PH, and VFMSUBADD231PH:
__m128h _mm_fmsubadd_ph (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fmsubadd_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fmsubadd_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fmsubadd_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h _mm256_fmsubadd_ph (__m256h a, __m256h b, __m256h c);
__m256h _mm256_mask_fmsubadd_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h _mm256_mask3_fmsubadd_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h _mm256_maskz_fmsubadd_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h _mm512_fmsubadd_ph (__m512h a, __m512h b, __m512h c);
__m512h _mm512_mask_fmsubadd_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h _mm512_mask3_fmsubadd_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h _mm512_maskz_fmsubadd_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h _mm512_fmsubadd_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask_fmsubadd_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask3_fmsubadd_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h _mm512_maskz_fmsubadd_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

VFMSUBADD132PS/VFMSUBADD213PS/VFMSUBADD231PS—Fused Multiply-Alternating Subtract/Add of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 97 /r VFMSUBADD132PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm1 and xmm3/mem, subtract/add elements in xmm2 and put result in xmm1.
VEX.128.66.0F38.W0 A7 /r VFMSUBADD213PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/mem and put result in xmm1.
VEX.128.66.0F38.W0 B7 /r VFMSUBADD231PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm2 and xmm3/mem, subtract/add elements in xmm1 and put result in xmm1.
VEX.256.66.0F38.W0 97 /r VFMSUBADD132PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm1 and ymm3/mem, subtract/add elements in ymm2 and put result in ymm1.
VEX.256.66.0F38.W0 A7 /r VFMSUBADD213PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/mem and put result in ymm1.
VEX.256.66.0F38.W0 B7 /r VFMSUBADD231PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm2 and ymm3/mem, subtract/add elements in ymm1 and put result in ymm1.
EVEX.128.66.0F38.W0 97 /r VFMSUBADD132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm1 and xmm3/m128/m32bcst, subtract/add elements in xmm2 and put result in xmm1 subject to writemask k1.
EVEX.128.66.0F38.W0 A7 /r VFMSUBADD213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/m128/m32bcst and put result in xmm1 subject to writemask k1.
EVEX.128.66.0F38.W0 B7 /r VFMSUBADD231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm2 and xmm3/m128/m32bcst, subtract/add elements in xmm1 and put result in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 97 /r VFMSUBADD132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm1 and ymm3/m256/m32bcst, subtract/add elements in ymm2 and put result in ymm1 subject to writemask k1.
EVEX.256.66.0F38.W0 A7 /r VFMSUBADD213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/m256/m32bcst and put result in ymm1 subject to writemask k1.
EVEX.256.66.0F38.W0 B7 /r VFMSUBADD231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm2 and ymm3/m256/m32bcst, subtract/add elements in ymm1 and put result in ymm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 97 /r VFMSUBADD132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm1 and zmm3/m512/m32bcst, subtract/add elements in zmm2 and put result in zmm1 subject to writemask k1.
EVEX.512.66.0F38.W0 A7 /r VFMSUBADD213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm1 and zmm2, subtract/add elements in zmm3/m512/m32bcst and put result in zmm1 subject to writemask k1.
EVEX.512.66.0F38.W0 B7 /r VFMSUBADD231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm2 and zmm3/m512/m32bcst, subtract/add elements in zmm1 and put result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFMSUBADD132PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the first source operand to the corresponding packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single precision floating-point elements and adds the even single precision floating-point values in the second source operand, performs rounding and stores the resulting packed single precision floating-point values to the destination operand (first source operand).

VFMSUBADD213PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the corresponding packed single precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd single precision floating-point elements and adds the even single precision floating-point values in the third source operand, performs rounding and stores the resulting packed single precision floating-point values to the destination operand (first source operand).

VFMSUBADD231PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the corresponding packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single precision floating-point elements and adds the even single precision floating-point values in the first source operand, performs rounding and stores the resulting packed single precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMSUBADD132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM -1{
    n := 64*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
    DEST[n+63:n+32] := RoundFPControl_MXCSR(DEST[n+63:n+32]*SRC3[n+63:n+32] -SRC2[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUBADD213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM -1{
    n := 64*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] +SRC3[n+31:n])
    DEST[n+63:n+32] := RoundFPControl_MXCSR(SRC2[n+63:n+32]*DEST[n+63:n+32] -SRC3[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUBADD231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM -1{
    n := 64*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
    DEST[n+63:n+32] := RoundFPControl_MXCSR(SRC2[n+63:n+32]*SRC3[n+63:n+32] -DEST[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUBADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] :=
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
                ELSE DEST[i+31:i] :=
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                        RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[i+31:i])
                        ELSE
                            DEST[i+31:i] :=
                        RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] - SRC2[i+31:i])
                        ELSE
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
```

```
                    FI;
            FI

        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
                ELSE DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[31:0])
```

```
                    ELSE
                        DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
                FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
                        ELSE
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[31:0])
                    FI;
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
                ELSE DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
```

```
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                        RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
                        ELSE
                            DEST[i+31:i] :=
                        RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
                        ELSE
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
                    FI;
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBADDxxxPS __m512 _mm512_fmsubadd_ps(__m512 a, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 _mm512_fmsubadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512 _mm512_mask_fmsubadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 _mm512_maskz_fmsubadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 _mm512_mask3_fmsubadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMSUBADDxxxPS __m512 _mm512_mask_fmsubadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512 _mm512_maskz_fmsubadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512 _mm512_mask3_fmsubadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMSUBADDxxxPS __m256 _mm256_mask_fmsubadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMSUBADDxxxPS __m256 _mm256_maskz_fmsubadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMSUBADDxxxPS __m256 _mm256_mask3_fmsubadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMSUBADDxxxPS __m128 _mm_mask_fmsubadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBADDxxxPS __m128 _mm_maskz_fmsubadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBADDxxxPS __m128 _mm_mask3_fmsubadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBADDxxxPS __m128 _mm_fmsubadd_ps (__m128 a, __m128 b, __m128 c);
VFMSUBADDxxxPS __m256 _mm256_fmsubadd_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VFNMADD132PD/VFNMADD213PD/VFNMADD231PD—Fused Negative Multiply-Add of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 9C /r VFNMADD132PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.
VEX.128.66.0F38.W1 AC /r VFNMADD213PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.
VEX.128.66.0F38.W1 BC /r VFNMADD231PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.
VEX.256.66.0F38.W1 9C /r VFNMADD132PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and add to ymm2 and put result in ymm1.
VEX.256.66.0F38.W1 AC /r VFNMADD213PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/mem and put result in ymm1.
VEX.256.66.0F38.W1 BC /r VFNMADD231PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and add to ymm1 and put result in ymm1.
EVEX.128.66.0F38.W1 9C /r VFNMADD132PD xmm0 {k1}{z}, xmm1, xmm2/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm1 and xmm3/m128/m64bcst, negate the multiplication result and add to xmm2 and put result in xmm1.
EVEX.128.66.0F38.W1 AC /r VFNMADD213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/m128/m64bcst and put result in xmm1.
EVEX.128.66.0F38.W1 BC /r VFNMADD231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm2 and xmm3/m128/m64bcst, negate the multiplication result and add to xmm1 and put result in xmm1.
EVEX.256.66.0F38.W1 9C /r VFNMADD132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm1 and ymm3/m256/m64bcst, negate the multiplication result and add to ymm2 and put result in ymm1.
EVEX.256.66.0F38.W1 AC /r VFNMADD213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/m256/m64bcst and put result in ymm1.
EVEX.256.66.0F38.W1 BC /r VFNMADD231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm2 and ymm3/m256/m64bcst, negate the multiplication result and add to ymm1 and put result in ymm1.

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 9C /r VFNMADD132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm1 and zmm3/m512/m64bcst, negate the multiplication result and add to zmm2 and put result in zmm1.
EVEX.512.66.0F38.W1 AC /r VFNMADD213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm1 and zmm2, negate the multiplication result and add to zmm3/m512/m64bcst and put result in zmm1.
EVEX.512.66.0F38.W1 BC /r VFNMADD231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm2 and zmm3/m512/m64bcst, negate the multiplication result and add to zmm1 and put result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFNMADD132PD: Multiplies the two, four or eight packed double precision floating-point values from the first source operand to the two, four or eight packed double precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the two, four or eight packed double precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFNMADD213PD: Multiplies the two, four or eight packed double precision floating-point values from the second source operand to the two, four or eight packed double precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the two, four or eight packed double precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFNMADD231PD: Multiplies the two, four or eight packed double precision floating-point values from the second source to the two, four or eight packed double precision floating-point values in the third source operand, the negated infinite precision intermediate result to the two, four or eight packed double precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFNMADD132PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(-(DEST[n+63:n]*SRC3[n+63:n]) + SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMADD213PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(-(SRC2[n+63:n]*DEST[n+63:n]) + SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMADD231PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(-(SRC2[n+63:n]*SRC3[n+63:n]) + DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(-(DEST[i+63:i]*SRC3[i+63:i]) + SRC2[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[63:0]) + SRC2[i+63:i])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[i+63:i]) + SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(-(SRC2[i+63:i]*DEST[i+63:i]) + SRC3[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) + SRC3[63:0])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) + SRC3[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(-(SRC2[i+63:i]*SRC3[i+63:i]) + DEST[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[63:0]) + DEST[i+63:i])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[i+63:i]) + DEST[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMADDxxxPD __m512d _mm512_fnmadd_pd(__m512d a, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_fnmadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_mask_fnmadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_maskz_fnmadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_mask3_fnmadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFNMADDxxxPD __m512d _mm512_mask_fnmadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_maskz_fnmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_mask3_fnmadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFNMADDxxxPD __m256d _mm256_mask_fnmadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFNMADDxxxPD __m256d _mm256_maskz_fnmadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFNMADDxxxPD __m256d _mm256_mask3_fnmadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFNMADDxxxPD __m128d _mm_mask_fnmadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMADDxxxPD __m128d _mm_maskz_fnmadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMADDxxxPD __m128d _mm_mask3_fnmadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMADDxxxPD __m128d _mm_fnmadd_pd (__m128d a, __m128d b, __m128d c);
VFNMADDxxxPD __m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, "Type 2 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

VFNMADD132PS/VFNMADD213PS/VFNMADD231PS—Fused Negative Multiply-Add of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 9C /r VFNMADD132PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.
VEX.128.66.0F38.W0 AC /r VFNMADD213PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.
VEX.128.66.0F38.W0 BC /r VFNMADD231PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.
VEX.256.66.0F38.W0 9C /r VFNMADD132PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and add to ymm2 and put result in ymm1.
VEX.256.66.0F38.W0 AC /r VFNMADD213PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/mem and put result in ymm1.
VEX.256.66.0F38.0 BC /r VFNMADD231PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and add to ymm1 and put result in ymm1.
EVEX.128.66.0F38.W0 9C /r VFNMADD132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm1 and xmm3/m128/m32bcst, negate the multiplication result and add to xmm2 and put result in xmm1.
EVEX.128.66.0F38.W0 AC /r VFNMADD213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/m128/m32bcst and put result in xmm1.
EVEX.128.66.0F38.W0 BC /r VFNMADD231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm2 and xmm3/m128/m32bcst, negate the multiplication result and add to xmm1 and put result in xmm1.
EVEX.256.66.0F38.W0 9C /r VFNMADD132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm1 and ymm3/m256/m32bcst, negate the multiplication result and add to ymm2 and put result in ymm1.
EVEX.256.66.0F38.W0 AC /r VFNMADD213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/m256/m32bcst and put result in ymm1.
EVEX.256.66.0F38.W0 BC /r VFNMADD231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm2 and ymm3/m256/m32bcst, negate the multiplication result and add to ymm1 and put result in ymm1.

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 9C /r VFNMADD132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from zmm1 and zmm3/m512/m32bcst, negate the multiplication result and add to zmm2 and put result in zmm1.
EVEX.512.66.0F38.W0 AC /r VFNMADD213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm1 and zmm2, negate the multiplication result and add to zmm3/m512/m32bcst and put result in zmm1.
EVEX.512.66.0F38.W0 BC /r VFNMADD231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm2 and zmm3/m512/m32bcst, negate the multiplication result and add to zmm1 and put result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFNMADD132PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the first source operand to the four, eight or sixteen packed single precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFNMADD213PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the four, eight or sixteen packed single precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single precision floating-point values in the third source operand, performs rounding and stores the resulting the four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFNMADD231PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the four, eight or sixteen packed single precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFNMADD132PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(- (DEST[n+31:n]*SRC3[n+31:n]) + SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMADD213PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(- (SRC2[n+31:n]*DEST[n+31:n]) + SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMADD231PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(- (SRC2[n+31:n]*SRC3[n+31:n]) + DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl(-(DEST[i+31:i]*SRC3[i+31:i]) + SRC2[i+31:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[31:0]) + SRC2[i+31:i])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[i+31:i]) + SRC2[i+31:i])
            FI;

        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[i+31:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[31:0])

                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl(-(SRC2[i+31:i]*SRC3[i+31:i]) + DEST[i+31:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[31:0]) + DEST[i+31:i])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) + DEST[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMADDxxxPS __m512 _mm512_fnmadd_ps(__m512 a, __m512 b, __m512 c);
VFNMADDxxxPS __m512 _mm512_fnmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m512 _mm512_mask_fnmadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFNMADDxxxPS __m512 _mm512_maskz_fnmadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFNMADDxxxPS __m512 _mm512_mask3_fnmadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFNMADDxxxPS __m512 _mm512_mask_fnmadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m512 _mm512_maskz_fnmadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m512 _mm512_mask3_fnmadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFNMADDxxxPS __m256 _mm256_mask_fnmadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFNMADDxxxPS __m256 _mm256_maskz_fnmadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFNMADDxxxPS __m256 _mm256_mask3_fnmadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFNMADDxxxPS __m128 _mm_mask_fnmadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMADDxxxPS __m128 _mm_maskz_fnmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMADDxxxPS __m128 _mm_mask3_fnmadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMADDxxxPS __m128 _mm_fnmadd_ps (__m128 a, __m128 b, __m128 c);
VFNMADDxxxPS __m256 _mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VFNMADD132SD/VFNMADD213SD/VFNMADD231SD—Fused Negative Multiply-Add of Scalar Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W1 9D /r VFNMADD132SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double precision floating-point value from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.
VEX.LIG.66.0F38.W1 AD /r VFNMADD213SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.
VEX.LIG.66.0F38.W1 BD /r VFNMADD231SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double precision floating-point value from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 9D /r VFNMADD132SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar double precision floating-point value from xmm1 and xmm3/m64, negate the multiplication result and add to xmm2 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 AD /r VFNMADD213SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar double precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m64 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 BD /r VFNMADD231SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar double precision floating-point value from xmm2 and xmm3/m64, negate the multiplication result and add to xmm1 and put result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

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 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations

involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFNMADD132SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(-(DEST[63:0]*SRC3[63:0]) + SRC2[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMADD213SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(-(SRC2[63:0]*DEST[63:0]) + SRC3[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMADD231SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(-(SRC2[63:0]*SRC3[63:0]) + DEST[63:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMADD132SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(- (DEST[63:0]*SRC3[63:0]) + SRC2[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMADD213SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(- (SRC2[63:0]*DEST[63:0]) + SRC3[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMADD231SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(- (SRC2[63:0]*SRC3[63:0]) + DEST[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMADDxxxSD __m128d _mm_fnmadd_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFNMADDxxxSD __m128d _mm_mask_fnmadd_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMADDxxxSD __m128d _mm_maskz_fnmadd_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMADDxxxSD __m128d _mm_mask3_fnmadd_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMADDxxxSD __m128d _mm_mask_fnmadd_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFNMADDxxxSD __m128d _mm_maskz_fnmadd_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFNMADDxxxSD __m128d _mm_mask3_fnmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFNMADDxxxSD __m128d _mm_fnmadd_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VFNMADD132SS/VFNMADD213SS/VFNMADD231SS—Fused Negative Multiply-Add of Scalar Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W0 9D /r VFNMADD132SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and add to xmm2 and put result in xmm1.
VEX.LIG.66.0F38.W0 AD /r VFNMADD213SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m32 and put result in xmm1.
VEX.LIG.66.0F38.W0 BD /r VFNMADD231SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and add to xmm1 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 9D /r VFNMADD132SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and add to xmm2 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 AD /r VFNMADD213SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m32 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 BD /r VFNMADD231SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and add to xmm1 and put result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

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 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations

involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFNMADD132SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(-(DEST[31:0]*SRC3[31:0]) + SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMADD213SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(-(SRC2[31:0]*DEST[31:0]) + SRC3[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMADD231SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(-(SRC2[31:0]*SRC3[63:0]) + DEST[31:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMADD132SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(- (DEST[31:0]*SRC3[31:0]) + SRC2[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMADD213SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(- (SRC2[31:0]*DEST[31:0]) + SRC3[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMADD231SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(- (SRC2[31:0]*SRC3[31:0]) + DEST[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMADDxxxSS __m128 _mm_fnmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 _mm_mask_fnmadd_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMADDxxxSS __m128 _mm_maskz_fnmadd_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMADDxxxSS __m128 _mm_mask3_fnmadd_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMADDxxxSS __m128 _mm_mask_fnmadd_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 _mm_maskz_fnmadd_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 _mm_mask3_fnmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNMADDxxxSS __m128 _mm_fnmadd_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VFNMSUB132PD/VFNMSUB213PD/VFNMSUB231PD—Fused Negative Multiply-Subtract of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 9E /r VFNMSUB132PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.
VEX.128.66.0F38.W1 AE /r VFNMSUB213PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.
VEX.128.66.0F38.W1 BE /r VFNMSUB231PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.
VEX.256.66.0F38.W1 9E /r VFNMSUB132PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and subtract ymm2 and put result in ymm1.
VEX.256.66.0F38.W1 AE /r VFNMSUB213PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/mem and put result in ymm1.
VEX.256.66.0F38.W1 BE /r VFNMSUB231PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and subtract ymm1 and put result in ymm1.
EVEX.128.66.0F38.W1 9E /r VFNMSUB132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm1 and xmm3/m128/m64bcst, negate the multiplication result and subtract xmm2 and put result in xmm1.
EVEX.128.66.0F38.W1 AE /r VFNMSUB213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m128/m64bcst and put result in xmm1.
EVEX.128.66.0F38.W1 BE /r VFNMSUB231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm2 and xmm3/m128/m64bcst, negate the multiplication result and subtract xmm1 and put result in xmm1.
EVEX.256.66.0F38.W1 9E /r VFNMSUB132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm1 and ymm3/m256/m64bcst, negate the multiplication result and subtract ymm2 and put result in ymm1.
EVEX.256.66.0F38.W1 AE /r VFNMSUB213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/m256/m64bcst and put result in ymm1.
EVEX.256.66.0F38.W1 BE /r VFNMSUB231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm2 and ymm3/m256/m64bcst, negate the multiplication result and subtract ymm1 and put result in ymm1.

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 9E /r VFNMSUB132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm1 and zmm3/m512/m64bcst, negate the multiplication result and subtract zmm2 and put result in zmm1.
EVEX.512.66.0F38.W1 AE /r VFNMSUB213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm1 and zmm2, negate the multiplication result and subtract zmm3/m512/m64bcst and put result in zmm1.
EVEX.512.66.0F38.W1 BE /r VFNMSUB231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm2 and zmm3/m512/m64bcst, negate the multiplication result and subtract zmm1 and put result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFNMSUB132PD: Multiplies the two, four or eight packed double precision floating-point values from the first source operand to the two, four or eight packed double precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFNMSUB213PD: Multiplies the two, four or eight packed double precision floating-point values from the second source operand to the two, four or eight packed double precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFNMSUB231PD: Multiplies the two, four or eight packed double precision floating-point values from the second source to the two, four or eight packed double precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFNMSUB132PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR( - (DEST[n+63:n]*SRC3[n+63:n]) - SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMSUB213PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR( - (SRC2[n+63:n]*DEST[n+63:n]) - SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMSUB231PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR( - (SRC2[n+63:n]*SRC3[n+63:n]) - DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(-(DEST[i+63:i]*SRC3[i+63:i]) - SRC2[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[63:0]) - SRC2[i+63:i])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[i+63:i]) - SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[63:0])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(-(SRC2[i+63:i]*SRC3[i+63:i]) - DEST[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[63:0]) - DEST[i+63:i])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[i+63:i]) - DEST[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUBxxxPD __m512d _mm512_fnmsub_pd(__m512d a, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d _mm512_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d _mm512_mask_fnmsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d _mm512_maskz_fnmsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d _mm512_mask3_fnmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFNMSUBxxxPD __m512d _mm512_mask_fnmsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d _mm512_maskz_fnmsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d _mm512_mask3_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFNMSUBxxxPD __m256d _mm256_mask_fnmsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFNMSUBxxxPD __m256d _mm256_maskz_fnmsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFNMSUBxxxPD __m256d _mm256_mask3_fnmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFNMSUBxxxPD __m128d _mm_mask_fnmsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMSUBxxxPD __m128d _mm_maskz_fnmsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMSUBxxxPD __m128d _mm_mask3_fnmsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMSUBxxxPD __m128d _mm_fnmsub_pd (__m128d a, __m128d b, __m128d c);
VFNMSUBxxxPD __m256d _mm256_fnmsub_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VFNMSUB132PS/VFNMSUB213PS/VFNMSUB231PS—Fused Negative Multiply-Subtract of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 9E /r VFNMSUB132PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.
VEX.128.66.0F38.W0 AE /r VFNMSUB213PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.
VEX.128.66.0F38.W0 BE /r VFNMSUB231PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.
VEX.256.66.0F38.W0 9E /r VFNMSUB132PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and subtract ymm2 and put result in ymm1.
VEX.256.66.0F38.W0 AE /r VFNMSUB213PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/mem and put result in ymm1.
VEX.256.66.0F38.0 BE /r VFNMSUB231PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and subtract ymm1 and put result in ymm1.
EVEX.128.66.0F38.W0 9E /r VFNMSUB132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, negate the multiplication result and subtract xmm2 and put result in xmm1.
EVEX.128.66.0F38.W0 AE /r VFNMSUB213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m128/m32bcst and put result in xmm1.
EVEX.128.66.0F38.W0 BE /r VFNMSUB231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, negate the multiplication result subtract add to xmm1 and put result in xmm1.
EVEX.256.66.0F38.W0 9E /r VFNMSUB132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, negate the multiplication result and subtract ymm2 and put result in ymm1.
EVEX.256.66.0F38.W0 AE /r VFNMSUB213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/m256/m32bcst and put result in ymm1.
EVEX.256.66.0F38.W0 BE /r VFNMSUB231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, negate the multiplication result subtract add to ymm1 and put result in ymm1.

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 9E /r VFNMSUB132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, negate the multiplication result and subtract zmm2 and put result in zmm1.
EVEX.512.66.0F38.W0 AE /r VFNMSUB213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single-precision floating-point values from zmm1 and zmm2, negate the multiplication result and subtract zmm3/m512/m32bcst and put result in zmm1.
EVEX.512.66.0F38.W0 BE /r VFNMSUB231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, negate the multiplication result subtract add to zmm1 and put result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFNMSUB132PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the first source operand to the four, eight or sixteen packed single precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFNMSUB213PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the four, eight or sixteen packed single precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFNMSUB231PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source to the four, eight or sixteen packed single precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFNMSUB132PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR( - (DEST[n+31:n]*SRC3[n+31:n]) - SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMSUB213PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR( - (SRC2[n+31:n]*DEST[n+31:n]) - SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMSUB231PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR( - (SRC2[n+31:n]*SRC3[n+31:n]) - DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl(-(DEST[i+31:i]*SRC3[i+31:i]) - SRC2[i+31:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[31:0]) - SRC2[i+31:i])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[i+31:i]) - SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[i+31:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[31:0])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) - DEST[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[31:0]) - DEST[i+31:i])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) - DEST[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUBxxxPS __m512 _mm512_fnmsub_ps(__m512 a, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 _mm512_fnmsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 _mm512_mask_fnmsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 _mm512_maskz_fnmsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 _mm512_mask3_fnmsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFNMSUBxxxPS __m512 _mm512_mask_fnmsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 _mm512_maskz_fnmsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 _mm512_mask3_fnmsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFNMSUBxxxPS __m256 _mm256_mask_fnmsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFNMSUBxxxPS __m256 _mm256_maskz_fnmsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFNMSUBxxxPS __m256 _mm256_mask3_fnmsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFNMSUBxxxPS __m128 _mm_mask_fnmsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMSUBxxxPS __m128 _mm_maskz_fnmsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMSUBxxxPS __m128 _mm_mask3_fnmsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMSUBxxxPS __m128 _mm_fnmsub_ps (__m128 a, __m128 b, __m128 c);
VFNMSUBxxxPS __m256 _mm256_fnmsub_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VFNMSUB132SD/VFNMSUB213SD/VFNMSUB231SD—Fused Negative Multiply-Subtract of Scalar Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W1 9F /r VFNMSUB132SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double precision floating-point value from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.
VEX.LIG.66.0F38.W1 AF /r VFNMSUB213SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.
VEX.LIG.66.0F38.W1 BF /r VFNMSUB231SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double precision floating-point value from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 9F /r VFNMSUB132SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar double precision floating-point value from xmm1 and xmm3/m64, negate the multiplication result and subtract xmm2 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 AF /r VFNMSUB213SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar double precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m64 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 BF /r VFNMSUB231SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar double precision floating-point value from xmm2 and xmm3/m64, negate the multiplication result and subtract xmm1 and put result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

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 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations

involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFNMSUB132SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(-(DEST[63:0]*SRC3[63:0]) - SRC2[63:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMSUB213SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(-(SRC2[63:0]*DEST[63:0]) - SRC3[63:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(-(SRC2[63:0]*SRC3[63:0]) - DEST[63:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMSUB132SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(- (DEST[63:0]*SRC3[63:0]) - SRC2[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMSUB213SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(- (SRC2[63:0]*DEST[63:0]) - SRC3[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMSUB231SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(- (SRC2[63:0]*SRC3[63:0]) - DEST[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUBxxxSD __m128d _mm_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFNMSUBxxxSD __m128d _mm_mask_fnmsub_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMSUBxxxSD __m128d _mm_maskz_fnmsub_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMSUBxxxSD __m128d _mm_mask3_fnmsub_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMSUBxxxSD __m128d _mm_mask_fnmsub_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFNMSUBxxxSD __m128d _mm_maskz_fnmsub_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFNMSUBxxxSD __m128d _mm_mask3_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFNMSUBxxxSD __m128d _mm_fnmsub_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VFNMSUB132SS/VFNMSUB213SS/VFNMSUB231SS—Fused Negative Multiply-Subtract of Scalar Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W0 9F /r VFNMSUB132SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and subtract xmm2 and put result in xmm1.
VEX.LIG.66.0F38.W0 AF /r VFNMSUB213SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m32 and put result in xmm1.
VEX.LIG.66.0F38.W0 BF /r VFNMSUB231SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and subtract xmm1 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 9F /r VFNMSUB132SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and subtract xmm2 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 AF /r VFNMSUB213SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m32 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 BF /r VFNMSUB231SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and subtract xmm1 and put result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

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 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations

involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFNMSUB132SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(-(DEST[31:0]*SRC3[31:0]) - SRC2[31:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMSUB213SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(-(SRC2[31:0]*DEST[31:0]) - SRC3[31:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMSUB231SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(-(SRC2[31:0]*SRC3[63:0]) - DEST[31:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMSUB132SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(- (DEST[31:0]*SRC3[31:0]) - SRC2[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMSUB213SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(- (SRC2[31:0]*DEST[31:0]) - SRC3[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMSUB231SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(- (SRC2[31:0]*SRC3[31:0]) - DEST[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUBxxxSS __m128 _mm_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFNMSUBxxxSS __m128 _mm_mask_fnmsub_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMSUBxxxSS __m128 _mm_maskz_fnmsub_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMSUBxxxSS __m128 _mm_mask3_fnmsub_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMSUBxxxSS __m128 _mm_mask_fnmsub_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFNMSUBxxxSS __m128 _mm_maskz_fnmsub_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFNMSUBxxxSS __m128 _mm_mask3_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNMSUBxxxSS __m128 _mm_fnmsub_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VFPCLASSPD—Tests Types of Packed Float64 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 66 /r ib VFPCLASSPD k2 {k1}, xmm2/m128/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.256.66.0F3A.W1 66 /r ib VFPCLASSPD k2 {k1}, ymm2/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.512.66.0F3A.W1 66 /r ib VFPCLASSPD k2 {k1}, zmm2/m512/m64bcst, imm8	A	V/V	AVX512DQ OR AVX10.1	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

The FPCLASSPD instruction checks the packed double precision floating-point values for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result of each element is written to the corresponding bit in a mask register k2 according to the writemask k1. Bits [MAX_KL-1:8/4/2] of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-11.

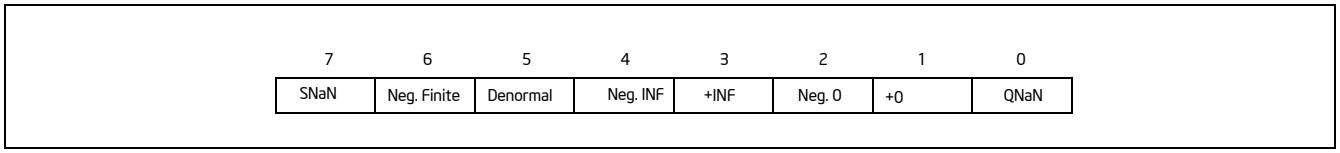


Figure 5-13. Imm8 Byte Specifier of Special Case Floating-Point Values for VFPCLASSPD/SD/PS/SS

Table 5-11. Classifier Operations for VFPCLASSPD/SD/PS/SS

Bits	Imm8[0]	Imm8[1]	Imm8[2]	Imm8[3]	Imm8[4]	Imm8[5]	Imm8[6]	Imm8[7]
Category	QNAN	PosZero	NegZero	PosINF	NegINF	Denormal	Negative	SNAN
Classifier	Checks for QNaN	Checks for +0	Checks for -0	Checks for +INF	Checks for -INF	Checks for Denormal	Checks for Negative finite	Checks for SNaN

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

```
CheckFPClassDP (tsrc[63:0], imm8[7:0]){

    //* Start checking the source operand for special type *//
    NegNum  := tsrc[63];
    IF (tsrc[62:52]=07FFh) Then ExpAllOnes := 1; FI;
    IF (tsrc[62:52]=0h) Then ExpAllZeros := 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
        MantAllZeros := 1;
    ELSIF (tsrc[51:0]=0h) Then
        MantAllZeros := 1;
    FI;
    ZeroNumber := ExpAllZeros AND MantAllZeros
    SignalingBit := tsrc[51];

    sNaN_res := ExpAllOnes AND  NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res := ExpAllOnes AND  NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res := NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
    Nzero_res := NegNum AND ExpAllZeros AND MantAllZeros; // -0
    PInf_res := NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
    NInf_res := NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
    Denorm_res := ExpAllZeros AND  NOT(MantAllZeros); // denorm
    FinNeg_res := NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

    bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR
              ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND PInf_res ) OR
              ( imm8[4] AND NInf_res ) OR ( imm8[5] AND Denorm_res ) OR
              ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
    Return bResult;
} //* end of CheckFPClassDP() *//
```

VFPCLASSPD (EVEX Encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1) AND (SRC *is memory*)
                THEN
                    DEST[j] := CheckFPClassDP(SRC1[63:0], imm8[7:0]);
                ELSE
                    DEST[j] := CheckFPClassDP(SRC1[i+63:i], imm8[7:0]);
            FI;
        ELSE DEST[j] := 0                    ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFPCLASSPD __mmask8 _mm512_fpclass_pd_mask(__m512d a, int c);
VFPCLASSPD __mmask8 _mm512_mask_fpclass_pd_mask(__mmask8 m, __m512d a, int c)
VFPCLASSPD __mmask8 _mm256_fpclass_pd_mask(__m256d a, int c)
VFPCLASSPD __mmask8 _mm256_mask_fpclass_pd_mask(__mmask8 m, __m256d a, int c)
VFPCLASSPD __mmask8 _mm_fpclass_pd_mask(__m128d a, int c)
VFPCLASSPD __mmask8 _mm_mask_fpclass_pd_mask(__mmask8 m, __m128d a, int c)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, "Type E4 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VFPCLASSPH—Test Types of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.0F3A.W0 66 /r /ib VFPCLASSPH k1{k2}, xmm1/m128/m16bcst, imm8	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Test the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.256.NP.0F3A.W0 66 /r /ib VFPCLASSPH k1{k2}, ymm1/m256/m16bcst, imm8	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Test the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.512.NP.0F3A.W0 66 /r /ib VFPCLASSPH k1{k2}, zmm1/m512/m16bcst, imm8	A	V/V	AVX512_FP16 OR AVX10.1	Test the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8 (r)	N/A

Description

This instruction checks the packed FP16 values in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against; see Table 5-12 for the categories. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the corresponding bits in the destination mask register according to the writemask.

Table 5-12. Classifier Operations for VFPCLASSPH/VFPCLASSSH

Bits	Category	Classifier
imm8[0]	QNAN	Checks for QNAN
imm8[1]	PosZero	Checks +0
imm8[2]	NegZero	Checks for -0
imm8[3]	PosINF	Checks for $+\infty$
imm8[4]	NegINF	Checks for $-\infty$
imm8[5]	Denormal	Checks for Denormal
imm8[6]	Negative	Checks for Negative finite
imm8[7]	SNAN	Checks for SNAN

Operation

```
def check_fp_class_fp16(tsrc, imm8):
    negative := tsrc[15]
    exponent_all_ones := (tsrc[14:10] == 0x1F)
    exponent_all_zeros := (tsrc[14:10] == 0)
    mantissa_all_zeros := (tsrc[9:0] == 0)
    zero := exponent_all_zeros and mantissa_all_zeros
    signaling_bit := tsrc[9]

    snan := exponent_all_ones and not(mantissa_all_zeros) and not(signaling_bit)
    qnan := exponent_all_ones and not(mantissa_all_zeros) and signaling_bit
    positive_zero := not(negative) and zero
    negative_zero := negative and zero
    positive_infinity := not(negative) and exponent_all_ones and mantissa_all_zeros
    negative_infinity := negative and exponent_all_ones and mantissa_all_zeros
    denormal := exponent_all_zeros and not(mantissa_all_zeros)
    finite_negative := negative and not(exponent_all_ones) and not(zero)

    return (imm8[0] and qnan) OR
        (imm8[1] and positive_zero) OR
        (imm8[2] and negative_zero) OR
        (imm8[3] and positive_infinity) OR
        (imm8[4] and negative_infinity) OR
        (imm8[5] and denormal) OR
        (imm8[6] and finite_negative) OR
        (imm8[7] and snan)
```

VFPCLASSPH dest{k2}, src, imm8

```
VL = 128, 256 or 512
KL := VL/16

FOR i := 0 to KL-1:
    IF k2[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := SRC.fp16[0]
        ELSE:
            tsrc := SRC.fp16[i]
        DEST.bit[i] := check_fp_class_fp16(tsrc, imm8)
    ELSE:
        DEST.bit[i] := 0

DEST[MAXKL-1:kl] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFPCLASSPH __mmask8 _mm_fpclass_ph_mask (__m128h a, int imm8);
VFPCLASSPH __mmask8 _mm_mask_fpclass_ph_mask (__mmask8 k1, __m128h a, int imm8);
VFPCLASSPH __mmask16 _mm256_fpclass_ph_mask (__m256h a, int imm8);
VFPCLASSPH __mmask16 _mm256_mask_fpclass_ph_mask (__mmask16 k1, __m256h a, int imm8);
VFPCLASSPH __mmask32 _mm512_fpclass_ph_mask (__m512h a, int imm8);
VFPCLASSPH __mmask32 _mm512_mask_fpclass_ph_mask (__mmask32 k1, __m512h a, int imm8);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instructions, see Table 2-51, “Type E4 Class Exception Conditions.”

VFPCLASSPS—Tests Types of Packed Float32 Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 66 /r ib VFPCLASSPS k2 {k1}, xmm2/m128/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.256.66.0F3A.W0 66 /r ib VFPCLASSPS k2 {k1}, ymm2/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.512.66.0F3A.W0 66 /r ib VFPCLASSPS k2 {k1}, zmm2/m512/m32bcst, imm8	A	V/V	AVX512DQ OR AVX10.1	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

The FPCLASSPS instruction checks the packed single precision floating-point values for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result of each element is written to the corresponding bit in a mask register k2 according to the writemask k1. Bits [MAX_KL-1:16/8/4] of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-11.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

```
CheckFPClassSP (tsrc[31:0], imm8[7:0]){

    //* Start checking the source operand for special type *//
    NegNum  := tsrc[31];
    IF (tsrc[30:23]=0FFh) Then ExpAllOnes := 1; FI;
    IF (tsrc[30:23]=0h) Then ExpAllZeros := 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
          MantAllZeros := 1;
    ELSIF (tsrc[22:0]=0h) Then
          MantAllZeros := 1;
    FI;
    ZeroNumber= ExpAllZeros AND MantAllZeros
    SignalingBit= tsrc[22];
```

```
    sNaN_res := ExpAllOnes AND  NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res := ExpAllOnes AND  NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res := NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
    Nzero_res := NegNum AND ExpAllZeros AND MantAllZeros; // -0
    PInf_res := NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
    NInf_res := NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
    Denorm_res := ExpAllZeros AND  NOT(MantAllZeros); // denorm
    FinNeg_res := NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

    bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR
              ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND PInf_res ) OR
              ( imm8[4] AND NInf_res ) OR ( imm8[5] AND Denorm_res ) OR
              ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
    Return bResult;
} //* end of CheckSPClassSP() *//
```

VFPCLASSPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1) AND (SRC *is memory*)
                THEN
                    DEST[j] := CheckFPClassDP(SRC1[31:0], imm8[7:0]);
                ELSE
                    DEST[j] := CheckFPClassDP(SRC1[i+31:i], imm8[7:0]);
            FI;
        ELSE  DEST[j] := 0                    ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFPCLASSPS __mmask16 _mm512_fpclass_ps_mask(__m512 a, int c);
VFPCLASSPS __mmask16 _mm512_mask_fpclass_ps_mask(__mmask16 m, __m512 a, int c)
VFPCLASSPS __mmask8 _mm256_fpclass_ps_mask(__m256 a, int c)
VFPCLASSPS __mmask8 _mm256_mask_fpclass_ps_mask(__mmask8 m, __m256 a, int c)
VFPCLASSPS __mmask8 _mm_fpclass_ps_mask(__m128 a, int c)
VFPCLASSPS __mmask8 _mm_mask_fpclass_ps_mask(__mmask8 m, __m128 a, int c)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, "Type E4 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VFPCLASSSD—Tests Type of a Scalar Float64 Value

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W1 67 /r ib VFPCLASSSD k2 {k1}, xmm2/m64, imm8	A	V/V	AVX512DQ OR AVX10.1	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

The FPCLASSSD instruction checks the low double precision floating-point value in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the low bit in a mask register k2 according to the writemask k1. Bits MAX_KL-1: 1 of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-11.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

```
CheckFPClassDP (tsrc[63:0], imm8[7:0]){

    NegNum  := tsrc[63];
    IF (tsrc[62:52]=07FFh) Then ExpAllOnes := 1; FI;
    IF (tsrc[62:52]=0h) Then ExpAllZeros := 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
          MantAllZeros := 1;
    ELSIF (tsrc[51:0]=0h) Then
          MantAllZeros := 1;
    FI;
    ZeroNumber := ExpAllZeros AND MantAllZeros
    SignalingBit := tsrc[51];

    sNaN_res := ExpAllOnes AND  NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res := ExpAllOnes AND  NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res := NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
    Nzero_res := NegNum AND ExpAllZeros AND MantAllZeros; // -0
    PInf_res := NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
    NInf_res := NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
    Denorm_res := ExpAllZeros AND  NOT(MantAllZeros); // denorm
    FinNeg_res := NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

    bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR
              ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND PInf_res ) OR
              ( imm8[4] AND NInf_res ) OR ( imm8[5] AND Denorm_res ) OR
              ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
    Return bResult;
```

```
} //* end of CheckFPClassDP() *//
```

VFPCLASSSD (EVEX encoded version)

```
IF k1[0] OR *no writemask*
    THEN DEST[0] :=
                CheckFPClassDP(SRC1[63:0], imm8[7:0])
    ELSE  DEST[0] := 0                  ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFPCLASSSD __mmask8 _mm_fpclass_sd_mask(__m128d a, int c)
VFPCLASSSD __mmask8 _mm_mask_fpclass_sd_mask(__mmask8 m, __m128d a, int c)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-55, "Type E6 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VFPCLASSSH—Test Types of Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.NP.0F3A.W0 67 /r /ib VFPCLASSSH k1{k2}, xmm1/m16, imm8	A	V/V	AVX512_FP16 OR AVX10.1	Test the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	ModRM:r/m (r)	imm8 (r)	N/A

Description

This instruction checks the low FP16 value in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against; see Table 5-12 for the categories. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the low bit in the destination mask register according to the writemask. The other bits in the destination mask register are zeroed.

Operation

VFPCLASSSH dest{k2}, src, imm8

```
IF k2[0] or *no writemask*:
    DEST.bit[0] := check_fp_class_fp16(src.fp16[0], imm8)    // see VFPCLASSPH
ELSE:
    DEST.bit[0] := 0

DEST[MAXKL-1:1] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFPCLASSSH __mmask8 _mm_fpclass_sh_mask (__m128h a, int imm8);
VFPCLASSSH __mmask8 _mm_mask_fpclass_sh_mask (__mmask8 k1, __m128h a, int imm8);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instructions, see Table 2-60, "Type E10 Class Exception Conditions."

VFPCLASSSS—Tests Type of a Scalar Float32 Value

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W0 67 /r VFPCLASSSS k2 {k1}, xmm2/m32, imm8	A	V/V	AVX512DQ OR AVX10.1	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

The FPCLASSSS instruction checks the low single precision floating-point value in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the low bit in a mask register k2 according to the writemask k1. Bits MAX_KL-1: 1 of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-11.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

```
CheckFPClassSP (tsrc[31:0], imm8[7:0]){

    //* Start checking the source operand for special type *//
    NegNum  := tsrc[31];
    IF (tsrc[30:23]=0FFh) Then ExpAllOnes := 1; FI;
    IF (tsrc[30:23]=0h) Then ExpAllZeros := 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
        MantAllZeros := 1;
    ELSIF (tsrc[22:0]=0h) Then
        MantAllZeros := 1;
    FI;
    ZeroNumber= ExpAllZeros AND MantAllZeros
    SignalingBit= tsrc[22];

    sNaN_res := ExpAllOnes AND  NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res := ExpAllOnes AND  NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res := NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
    Nzero_res := NegNum AND ExpAllZeros AND MantAllZeros; // -0
    PInf_res := NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
    NInf_res := NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
    Denorm_res := ExpAllZeros AND  NOT(MantAllZeros); // denorm
    FinNeg_res := NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

    bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR
              ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND PInf_res ) OR
              ( imm8[4] AND NInf_res ) OR ( imm8[5] AND Denorm_res ) OR
              ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
```

```
    Return bResult;
} //* end of CheckSPClassSP() *//
```

VFPCLASSSS (EVEX encoded version)

```
IF k1[0] OR *no writemask*
    THEN DEST[0] :=
            CheckFPClassSP(SRC1[31:0], imm8[7:0])
    ELSE  DEST[0] := 0                  ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFPCLASSSS __mmask8 _mm_fpclass_ss_mask(__m128 a, int c)
VFPCLASSSS __mmask8 _mm_mask_fpclass_ss_mask(__mmask8 m, __m128 a, int c)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-55, "Type E6 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VGATHERDPD/VGATHERQPD—Gather Packed Double Precision Floating-Point Values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 92 /r VGATHERDPD xmm1, vm32x, xmm2	RMV	V/V	AVX2	Using dword indices specified in vm32x, gather double precision floating-point values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.
VEX.128.66.0F38.W1 93 /r VGATHERQPD xmm1, vm64x, xmm2	RMV	V/V	AVX2	Using qword indices specified in vm64x, gather double precision floating-point values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.
VEX.256.66.0F38.W1 92 /r VGATHERDPD ymm1, vm32x, ymm2	RMV	V/V	AVX2	Using dword indices specified in vm32x, gather double precision floating-point values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.
VEX.256.66.0F38.W1 93 /r VGATHERQPD ymm1, vm64y, ymm2	RMV	V/V	AVX2	Using qword indices specified in vm64y, gather double precision floating-point values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	N/A

Description

The instruction conditionally loads up to 2 or 4 double precision floating-point values from memory addresses specified by the memory operand (the second operand) and using qword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using dword indices in the lower half of the mask register, the instruction conditionally loads up to 2 or 4 double precision floating-point values from the VSIB addressing memory operand, and updates the destination register.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: The instruction will gather two double precision floating-point values. For dword indices, only the lower two indices in the vector index register are used.

VEX.256 version: The instruction will gather four double precision floating-point values. For dword indices, only the lower four indices in the vector index register are used.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a #UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

```
DEST := SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB:[7:6];
DISP: optional 1, 4 byte displacement;
MASK := SRC3;

VGATHERDPD (VEX.128 version)
MASK[MAXVL-1:128] := 0;
FOR j := 0 to 1
    i := j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] := FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] := 0;
    FI;
ENDFOR
FOR j := 0 to 1
    k := j * 32;
    i := j * 64;
    DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX[k+31:k])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] := FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63: i] := 0;
ENDFOR
DEST[MAXVL-1:128] := 0;
```

VGATHERQPD (VEX.128 version)

```
MASK[MAXVL-1:128] := 0;
FOR j := 0 to 1
    i := j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] := FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] := 0;
    FI;
ENDFOR
FOR j := 0 to 1
    i := j * 64;
    DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] := FETCH_64BITS(DATA_ADDR); // a fault exits this instruction
    FI;
    MASK[i +63: i] := 0;
ENDFOR
DEST[MAXVL-1:128] := 0;
```

VGATHERQPD (VEX.256 version)

```
MASK[MAXVL-1:256] := 0;
FOR j := 0 to 3
    i := j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] := FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] := 0;
    FI;
ENDFOR
FOR j := 0 to 3
    i := j * 64;
    DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] := FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63: i] := 0;
ENDFOR
DEST[MAXVL-1:256] := 0;
```

VGATHERDPD (VEX.256 version)

```
MASK[MAXVL-1:256] := 0;
FOR j := 0 to 3
    i := j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] := FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] := 0;
    FI;
ENDFOR
FOR j := 0 to 3
    k := j * 32;
    i := j * 64;
    DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[k+31:k])*SCALE + DISP;
```

```
    IF MASK[63+i] THEN
        DEST[i +63:i] := FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63:i] := 0;
ENDFOR
DEST[MAXVL-1:256] := 0;
```

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 Table 2-27, "Type 12 Class Exception Conditions."

VGATHERDPS/VGATHERDPD—Gather Packed Single, Packed Double with Signed Dword Indices

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 92 /vsib VGATHERDPS xmm1 {k1}, vm32x	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.256.66.0F38.W0 92 /vsib VGATHERDPS ymm1 {k1}, vm32y	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.512.66.0F38.W0 92 /vsib VGATHERDPS zmm1 {k1}, vm32z	A	V/V	AVX512F OR AVX10.1	Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.128.66.0F38.W1 92 /vsib VGATHERDPD xmm1 {k1}, vm32x	A	V/V	(AVX512VL AND AVX512F) R AVX10.1[1]	Using signed dword indices, gather float64 vector into float64 vector xmm1 using k1 as completion mask.
EVEX.256.66.0F38.W1 92 /vsib VGATHERDPD ymm1 {k1}, vm32x	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, gather float64 vector into float64 vector ymm1 using k1 as completion mask.
EVEX.512.66.0F38.W1 92 /vsib VGATHERDPD zmm1 {k1}, vm32y	A	V/V	AVX512F OR AVX10.1	Using signed dword indices, gather float64 vector into float64 vector zmm1 using k1 as completion mask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	N/A	N/A

Description

A set of single precision/double precision faulting-point memory locations pointed by base address BASE_ADDR and index vector V_INDEX with scale SCALE are gathered. The result is written into a vector register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the right most one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.

- Elements may be gathered in any order, but faults must be delivered in a right-to left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a vector register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1 or 4 byte displacement

VGATHERDPS (EVEX encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j]
        THEN DEST[i+31:i] :=
            MEM[BASE_ADDR +
                    SignExtend(VINDEX[i+31:i]) * SCALE + DISP]
            k1[j] := 0
        ELSE *DEST[i+31:i] := remains unchanged*
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL] := 0
```

VGATHERDPD (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j]
        THEN DEST[i+63:i] := MEM[BASE_ADDR +
                    SignExtend(VINDEX[k+31:k]) * SCALE + DISP]
            k1[j] := 0
        ELSE *DEST[i+63:i] := remains unchanged*
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGATHERDPD __m512d _mm512_i32gather_pd(__m256i vdx, void * base, int scale);

VGATHERDPD __m512d _mm512_mask_i32gather_pd(__m512d s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERDPD __m256d _mm256_mmask_i32gather_pd(__m256d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERDPD __m128d _mm_mmask_i32gather_pd(__m128d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERDPS __m512 _mm512_i32gather_ps(__m512i vdx, void * base, int scale);
VGATHERDPS __m512 _mm512_mask_i32gather_ps(__m512 s, __mmask16 k, __m512i vdx, void * base, int scale);
VGATHERDPS __m256 _mm256_mmask_i32gather_ps(__m256 s, __mmask8 k, __m256i vdx, void * base, int scale);
GATHERDPS __m128 _mm_mmask_i32gather_ps(__m128 s, __mmask8 k, __m128i vdx, void * base, int scale);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, "Type E12 Class Exception Conditions."

VGATHERDPS/VGATHERQPS—Gather Packed Single Precision Floating-Point Values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 92 /r VGATHERDPS xmm1, vm32x, xmm2	A	V/V	AVX2	Using dword indices specified in vm32x, gather single precision floating-point values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.
VEX.128.66.0F38.W0 93 /r VGATHERQPS xmm1, vm64x, xmm2	A	V/V	AVX2	Using qword indices specified in vm64x, gather single precision floating-point values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.
VEX.256.66.0F38.W0 92 /r VGATHERDPS ymm1, vm32y, ymm2	A	V/V	AVX2	Using dword indices specified in vm32y, gather single precision floating-point values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.
VEX.256.66.0F38.W0 93 /r VGATHERQPS xmm1, vm64y, xmm2	A	V/V	AVX2	Using qword indices specified in vm64y, gather single precision floating-point 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)	N/A

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 zero 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 zero the upper 128 bits of the destination.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

```
DEST := SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB:[7:6];
DISP: optional 1, 4 byte displacement;
MASK := SRC3;

VGATHERDPS (VEX.128 version)
MASK[MAXVL-1:128] := 0;
FOR j := 0 to 3
    i := j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] := FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] := 0;
    FI;
ENDFOR
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[MAXVL-1:128] := 0;
```

VGATHERQPS (VEX.128 version)

```
MASK[MAXVL-1:64] := 0;
FOR j := 0 to 3
    i := j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] := FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] := 0;
    FI;
ENDFOR
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
DEST[MAXVL-1:64] := 0;
```

VGATHERDPS (VEX.256 version)

```
MASK[MAXVL-1:256] := 0;
FOR j := 0 to 7
    i := j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] := FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] := 0;
    FI;
ENDFOR
FOR j := 0 to 7
    i := j * 32;
    DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[i+31:i])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] := FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] := 0;
ENDFOR
DEST[MAXVL-1:256] := 0;
```

VGATHERQPS (VEX.256 version)

```
MASK[MAXVL-1:128] := 0;
FOR j := 0 to 7
    i := j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] := FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] := 0;
    FI;
ENDFOR
FOR j := 0 to 3
```

```
    k := j * 64;
    i := j * 32;
    DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] := FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] := 0;
ENDFOR
DEST[MAXVL-1:128] := 0;
```

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 Table 2-27, "Type 12 Class Exception Conditions."

VGATHERQPS/VGATHERQPD—Gather Packed Single, Packed Double with Signed Qword Indices

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 93 /vsib VGATHERQPS xmm1 {k1}, vm64x	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.256.66.0F38.W0 93 /vsib VGATHERQPS xmm1 {k1}, vm64y	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.512.66.0F38.W0 93 /vsib VGATHERQPS ymm1 {k1}, vm64z	A	V/V	AVX512F OR AVX10.1	Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.128.66.0F38.W1 93 /vsib VGATHERQPD xmm1 {k1}, vm64x	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, gather float64 vector into float64 vector xmm1 using k1 as completion mask.
EVEX.256.66.0F38.W1 93 /vsib VGATHERQPD ymm1 {k1}, vm64y	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, gather float64 vector into float64 vector ymm1 using k1 as completion mask.
EVEX.512.66.0F38.W1 93 /vsib VGATHERQPD zmm1 {k1}, vm64z	A	V/V	AVX512F OR AVX10.1	Using signed qword indices, gather float64 vector into float64 vector zmm1 using k1 as completion mask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	N/A	N/A

Description

A set of 8 single precision/double precision faulting-point memory locations pointed by base address BASE_ADDR and index vector V_INDEX with scale SCALE are gathered. The result is written into vector a register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.

- Elements may be gathered in any order, but faults must be delivered in a right-to left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1 or 4 byte displacement

VGATHERQPS (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1

    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP]
            k1[j] := 0
        ELSE *DEST[i+31:i] := remains unchanged*
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL/2] := 0
```

VGATHERQPD (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := MEM[BASE_ADDR + (VINDEX[i+63:i]) * SCALE + DISP]
            k1[j] := 0
        ELSE *DEST[i+63:i] := remains unchanged*
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGATHERQPD __m512d _mm512_i64gather_pd(__m512i vdx, void * base, int scale);

VGATHERQPD __m512d _mm512_mask_i64gather_pd(__m512d s, __mmask8 k, __m512i vdx, void * base, int scale);
VGATHERQPD __m256d _mm256_mask_i64gather_pd(__m256d s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERQPD __m128d _mm_mask_i64gather_pd(__m128d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERQPS __m256 _mm512_i64gather_ps(__m512i vdx, void * base, int scale);
VGATHERQPS __m256 _mm512_mask_i64gather_ps(__m256 s, __mmask16 k, __m512i vdx, void * base, int scale);
VGATHERQPS __m128 _mm256_mask_i64gather_ps(__m128 s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERQPS __m128 _mm_mask_i64gather_ps(__m128 s, __mmask8 k, __m128i vdx, void * base, int scale);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, "Type E12 Class Exception Conditions."

VGETEXPPD—Convert Exponents of Packed Double Precision Floating-Point Values to Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 42 /r VGETEXPPD xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert the exponent of packed double precision floating-point values in the source operand to double precision floating-point results representing unbiased integer exponents and stores the results in the destination register.
EVEX.256.66.0F38.W1 42 /r VGETEXPPD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert the exponent of packed double precision floating-point values in the source operand to double precision floating-point results representing unbiased integer exponents and stores the results in the destination register.
EVEX.512.66.0F38.W1 42 /r VGETEXPPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}	A	V/V	AVX512F OR AVX10.1	Convert the exponent of packed double precision floating-point values in the source operand to double precision floating-point results representing unbiased integer exponents and stores the results in the destination under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Extracts the biased exponents from the normalized double precision floating-point representation of each qword data element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to double precision floating-point value and written to the corresponding qword elements of the destination operand (the first operand) as double precision floating-point numbers.

The destination operand is a ZMM/YMM/XMM register and updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Each GETEXP operation converts the exponent value into a floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-13.

The formula is:

GETEXP(x) = floor($\log_2$(|x|))

Notation **floor(x)** stands for the greatest integer not exceeding real number x.

Table 5-13. VGETEXPPD/SD Special Cases

Input Operand	Result	Comments
src1 = NaN	QNaN(src1)	If (SRC = SNaN) then #IE If (SRC = denormal) then #DE
0 < |src1| < INF	floor($\log_2$(|src1|))	
| src1| = +INF	+INF	
| src1| = 0	-INF	

Operation

```
NormalizeExpTinyDPFP(SRC[63:0])
{
```

```
    // Jbit is the hidden integral bit of a floating-point number. In case of denormal number it has the value of ZERO.
    Src.Jbit := 0;
    Dst.exp := 1;
    Dst.fraction := SRC[51:0];
    WHILE(Src.Jbit = 0)
    {
        Src.Jbit := Dst.fraction[51];              // Get the fraction MSB
        Dst.fraction := Dst.fraction << 1 ;        // One bit shift left
        Dst.exp-- ;                // Decrement the exponent
    }
    Dst.fraction := 0;             // zero out fraction bits
    Dst.sign := 1;                 // Return negative sign
    TMP[63:0] := MXCSR.DAZ? 0 : (Dst.sign << 63) OR (Dst.exp << 52) OR (Dst.fraction) ;
    Return (TMP[63:0]);
}

ConvertExpDPFP(SRC[63:0])
{
    Src.sign := 0;                 // Zero out sign bit
    Src.exp := SRC[62:52];
    Src.fraction := SRC[51:0];
    // Check for NaN
    IF (SRC = NaN)
    {
        IF ( SRC = SNAN ) SET IE;
        Return QNAN(SRC);
    }
    // Check for +INF
    IF (Src = +INF) RETURN (Src);

    // check if zero operand
    IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) Return (-INF);
    }
    ELSE        // check if denormal operand (notice that MXCSR.DAZ = 0)
    {
        IF ((Src.exp = 0) AND (Src.fraction != 0))
        {
            TMP[63:0] := NormalizeExpTinyDPFP(SRC[63:0]) ;     // Get Normalized Exponent
            Set #DE
        }
        ELSE        // exponent value is correct
        {
            TMP[63:0] := (Src.sign << 63) OR (Src.exp << 52) OR (Src.fraction) ;
        }
        TMP := SAR(TMP, 52) ;          // Shift Arithmetic Right
        TMP := TMP – 1023;             // Subtract Bias
        Return CvtI2D(TMP);            // Convert INT to double precision floating-point number
    }
}
```

VGETEXPPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
```

```
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN
                    DEST[i+63:i] :=
            ConvertExpDPFP(SRC[63:0])
                ELSE
                    DEST[i+63:i] :=
            ConvertExpDPFP(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VGETEXPPD __m512d _mm512_getexp_pd(__m512d a);
VGETEXPPD __m512d _mm512_mask_getexp_pd(__m512d s, __mmask8 k, __m512d a);
VGETEXPPD __m512d _mm512_maskz_getexp_pd( __mmask8 k, __m512d a);
VGETEXPPD __m512d _mm512_getexp_round_pd(__m512d a, int sae);
VGETEXPPD __m512d _mm512_mask_getexp_round_pd(__m512d s, __mmask8 k, __m512d a, int sae);
VGETEXPPD __m512d _mm512_maskz_getexp_round_pd( __mmask8 k, __m512d a, int sae);
VGETEXPPD __m256d _mm256_getexp_pd(__m256d a);
VGETEXPPD __m256d _mm256_mask_getexp_pd(__m256d s, __mmask8 k, __m256d a);
VGETEXPPD __m256d _mm256_maskz_getexp_pd( __mmask8 k, __m256d a);
VGETEXPPD __m128d _mm_getexp_pd(__m128d a);
VGETEXPPD __m128d _mm_mask_getexp_pd(__m128d s, __mmask8 k, __m128d a);
VGETEXPPD __m128d _mm_maskz_getexp_pd( __mmask8 k, __m128d a);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

See Table 2-48, "Type E2 Class Exception Conditions."
Additionally:

#UD	If EVEX.vvvv != 1111B.

VGETEXPPH—Convert Exponents of Packed FP16 Values to FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP6.W0 42 /r VGETEXPPH xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert the exponent of FP16 values in the source operand to FP16 results representing unbiased integer exponents and stores the results in the destination register subject to writemask k1.
EVEX.256.66.MAP6.W0 42 /r VGETEXPPH ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Convert the exponent of FP16 values in the source operand to FP16 results representing unbiased integer exponents and stores the results in the destination register subject to writemask k1.
EVEX.512.66.MAP6.W0 42 /r VGETEXPPH zmm1{k1}{z}, zmm2/m512/m16bcst {sae}	A	V/V	AVX512_FP16 OR AVX10.1	Convert the exponent of FP16 values in the source operand to FP16 results representing unbiased integer exponents and stores the results in the destination register subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction extracts the biased exponents from the normalized FP16 representation of each word element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to an FP16 value and written to the corresponding word elements of the destination operand (the first operand) as FP16 numbers.

The destination elements are updated according to the writemask.

Each GETEXP operation converts the exponent value into a floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-8.

The formula is:

GETEXP(x) = floor($\log_2$(|x|))

Notation **floor(x)** stands for maximal integer not exceeding real number x.

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPH). Thus, the VGETEXPPH instruction does not require software to handle SIMD floating-point exceptions.

Table 5-14. VGETEXPPH/VGETEXPSH Special Cases

Input Operand	Result	Comments
src1 = NaN	QNaN(src1)	If (SRC = SNaN), then #IE. If (SRC = denormal), then #DE.
0 < |src1| < INF	floor($\log_2$(|src1|))	
| src1| = +INF	+INF	
| src1| = 0	-INF	

Operation

```
def normalize_exponent_tiny_fp16(src):
    jbit := 0
    // src & dst are FP16 numbers with sign(1b), exp(5b) and fraction (10b) fields
    dst.exp := 1                        // write bits 14:10
    dst.fraction := src.fraction // copy bits 9:0
    while jbit == 0:
        jbit := dst.fraction[9]         // msb of the fraction
        dst.fraction := dst.fraction << 1
        dst.exp := dst.exp - 1
    dst.fraction := 0
    return dst

def getexp_fp16(src):
    src.sign := 0                       // make positive
    exponent_all_ones := (src[14:10] == 0x1F)
    exponent_all_zeros := (src[14:10] == 0)
    mantissa_all_zeros := (src[9:0] == 0)
    zero := exponent_all_zeros and mantissa_all_zeros
    signaling_bit := src[9]

    nan := exponent_all_ones and not(mantissa_all_zeros)
    snan := nan and not(signaling_bit)
    qnan := nan and signaling_bit
    positive_infinity := not(negative) and exponent_all_ones and mantissa_all_zeros
    denormal := exponent_all_zeros and not(mantissa_all_zeros)

    if nan:
        if snan:
            MXCSR.IE := 1
        return qnan(src)                // convert snan to a qnan
    if positive_infinity:
        return src
    if zero:
        return -INF
    if denormal:
        tmp := normalize_exponent_tiny_fp16(src)
        MXCSR.DE := 1
    else:
        tmp := src
    tmp := SAR(tmp, 10)                 // shift arithmetic right
    tmp := tmp - 15                     // subtract bias
    return convert_integer_to_fp16(tmp)
```

VGETEXPPH dest{k1}, src

```
VL = 128, 256 or 512
KL := VL/16

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.fp16[0]
        ELSE:
            tsrc := src.fp16[i]
        DEST.fp16[i] := getexp_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[i] := 0
    //else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VGETEXPPH __m128h _mm_getexp_ph (__m128h a);
VGETEXPPH __m128h _mm_mask_getexp_ph (__m128h src, __mmask8 k, __m128h a);
VGETEXPPH __m128h _mm_maskz_getexp_ph (__mmask8 k, __m128h a);
VGETEXPPH __m256h _mm256_getexp_ph (__m256h a);
VGETEXPPH __m256h _mm256_mask_getexp_ph (__m256h src, __mmask16 k, __m256h a);
VGETEXPPH __m256h _mm256_maskz_getexp_ph (__mmask16 k, __m256h a);
VGETEXPPH __m512h _mm512_getexp_ph (__m512h a);
VGETEXPPH __m512h _mm512_mask_getexp_ph (__m512h src, __mmask32 k, __m512h a);
VGETEXPPH __m512h _mm512_maskz_getexp_ph (__mmask32 k, __m512h a);
VGETEXPPH __m512h _mm512_getexp_round_ph (__m512h a, const int sae);
VGETEXPPH __m512h _mm512_mask_getexp_round_ph (__m512h src, __mmask32 k, __m512h a, const int sae);
VGETEXPPH __m512h _mm512_maskz_getexp_round_ph (__mmask32 k, __m512h a, const int sae);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VGETEXPPS—Convert Exponents of Packed Single Precision Floating-Point Values to Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 42 /r VGETEXPPS xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert the exponent of packed single-precision floating-point values in the source operand to single-precision floating-point results representing unbiased integer exponents and stores the results in the destination register.
EVEX.256.66.0F38.W0 42 /r VGETEXPPS ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert the exponent of packed single-precision floating-point values in the source operand to single-precision floating-point results representing unbiased integer exponents and stores the results in the destination register.
EVEX.512.66.0F38.W0 42 /r VGETEXPPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}	A	V/V	AVX512F OR AVX10.1	Convert the exponent of packed single-precision floating-point values in the source operand to single-precision floating-point results representing unbiased integer exponents and stores the results in the destination register.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Extracts the biased exponents from the normalized single precision floating-point representation of each dword element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to single precision floating-point value and written to the corresponding dword elements of the destination operand (the first operand) as single precision floating-point numbers.

The destination operand is a ZMM/YMM/XMM register and updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Each GETEXP operation converts the exponent value into a floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-15.

The formula is:

GETEXP(x) = floor($\log_2$(|x|))

Notation **floor(x)** stands for maximal integer not exceeding real number x.

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPD). Thus VGETEXPxx instruction do not require software to handle SIMD floating-point exceptions.

Table 5-15. VGETEXPPS/SS Special Cases

Input Operand	Result	Comments
src1 = NaN	QNaN(src1)	If (SRC = SNaN) then #IE If (SRC = denormal) then #DE
0 < |src1| < INF	floor($\log_2$(|src1|))	
| src1| = +INF	+INF	
| src1| = 0	-INF	

Figure 5-14 illustrates the VGETEXPPS functionality on input values with normalized representation.

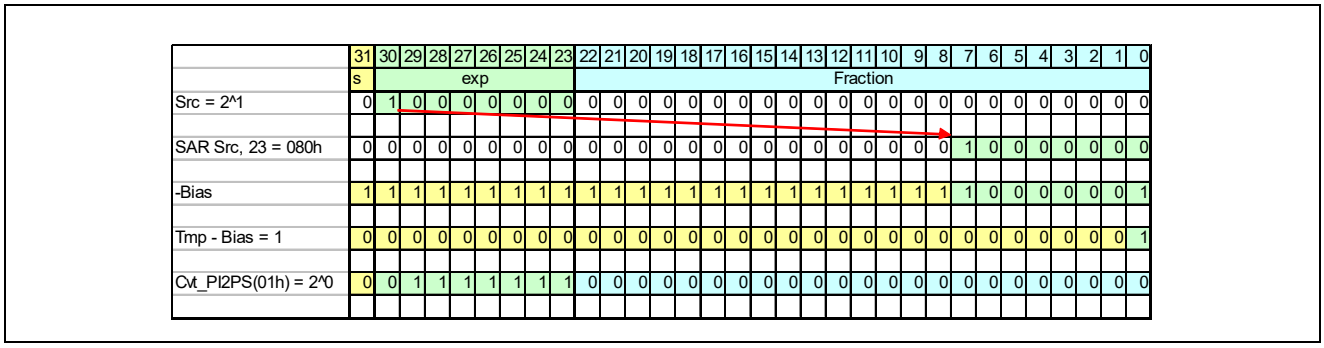


Figure 5-14. VGETEXPPS Functionality On Normal Input values

Operation

```
NormalizeExpTinySPFP(SRC[31:0])
{
    // Jbit is the hidden integral bit of a floating-point number. In case of denormal number it has the value of ZERO.
    Src.Jbit := 0;
    Dst.exp := 1;
    Dst.fraction := SRC[22:0];
    WHILE(Src.Jbit = 0)
    {
        Src.Jbit := Dst.fraction[22];          // Get the fraction MSB
        Dst.fraction := Dst.fraction << 1 ;    // One bit shift left
        Dst.exp-- ;                            // Decrement the exponent
    }
    Dst.fraction := 0;                         // zero out fraction bits
    Dst.sign := 1;                             // Return negative sign
    TMP[31:0] := MXCSR.DAZ? 0 : (Dst.sign << 31) OR (Dst.exp << 23) OR (Dst.fraction) ;
    Return (TMP[31:0]);
}
ConvertExpSPFP(SRC[31:0])
{
    Src.sign := 0;                             // Zero out sign bit
    Src.exp := SRC[30:23];
    Src.fraction := SRC[22:0];
    // Check for NaN
    IF (SRC = NaN)
    {
        IF ( SRC = SNAN ) SET IE;
```

```
        Return QNAN(SRC);
    }
    // Check for +INF
    IF (Src = +INF) RETURN (Src);

    // check if zero operand
    IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) Return (-INF);
    }
    ELSE            // check if denormal operand (notice that MXCSR.DAZ = 0)
    {
        IF ((Src.exp = 0) AND (Src.fraction != 0))
        {
            TMP[31:0] := NormalizeExpTinySPFP(SRC[31:0]) ;      // Get Normalized Exponent
            Set #DE
        }
        ELSE            // exponent value is correct
        {
            TMP[31:0] := (Src.sign << 31) OR (Src.exp << 23) OR (Src.fraction) ;
        }
        TMP := SAR(TMP, 23) ;           // Shift Arithmetic Right
        TMP := TMP – 127;               // Subtract Bias
        Return CvtI2S(TMP);             // Convert INT to single precision floating-point number
    }
}
```

VGETEXPPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN
                    DEST[i+31:i] :=
            ConvertExpSPFP(SRC[31:0])
                ELSE
                    DEST[i+31:i] :=
            ConvertExpSPFP(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPPS __m512 _mm512_getexp_ps(__m512 a);
VGETEXPPS __m512 _mm512_mask_getexp_ps(__m512 s, __mmask16 k, __m512 a);
VGETEXPPS __m512 _mm512_maskz_getexp_ps(__mmask16 k, __m512 a);
VGETEXPPS __m512 _mm512_getexp_round_ps(__m512 a, int sae);
VGETEXPPS __m512 _mm512_mask_getexp_round_ps(__m512 s, __mmask16 k, __m512 a, int sae);
VGETEXPPS __m512 _mm512_maskz_getexp_round_ps(__mmask16 k, __m512 a, int sae);
VGETEXPPS __m256 _mm256_getexp_ps(__m256 a);
VGETEXPPS __m256 _mm256_mask_getexp_ps(__m256 s, __mmask8 k, __m256 a);
VGETEXPPS __m256 _mm256_maskz_getexp_ps(__mmask8 k, __m256 a);
VGETEXPPS __m128 _mm_getexp_ps(__m128 a);
VGETEXPPS __m128 _mm_mask_getexp_ps(__m128 s, __mmask8 k, __m128 a);
VGETEXPPS __m128 _mm_maskz_getexp_ps(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

See Table 2-48, "Type E2 Class Exception Conditions."
Additionally:

#UD	If EVEX.vvvv != 1111B.

VGETEXPSD—Convert Exponents of Scalar Double Precision Floating-Point Value to Double Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W1 43 /r VGETEXPSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	A	V/V	AVX512F OR AVX10.1	Convert the biased exponent (bits 62:52) of the low double precision floating-point value in xmm3/m64 to a double precision floating-point value representing unbiased integer exponent. Stores the result to the low 64-bit of xmm1 under the writemask k1 and merge with the other elements of xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Extracts the biased exponent from the normalized double precision floating-point representation of the low qword data element of the source operand (the third operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. The integer value of the unbiased exponent is converted to double precision floating-point value and written to the destination operand (the first operand) as double precision floating-point numbers. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be a XMM register, the source operand can be a XMM register or a float64 memory location.

If writemasking is used, the low quadword element of the destination operand is conditionally updated depending on the value of writemask register k1. If writemasking is not used, the low quadword element of the destination operand is unconditionally updated.

Each GETEXP operation converts the exponent value into a floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-13.

The formula is:

$GETEXP(x) = floor(\log_2(|x|))$

Notation **floor(x)** stands for maximal integer not exceeding real number x.

Operation

```
// NormalizeExpTinyDPFP(SRC[63:0]) is defined in the Operation section of VGETEXPPD

// ConvertExpDPFP(SRC[63:0]) is defined in the Operation section of VGETEXPPD
```

VGETEXPSD (EVEX encoded version)

```
IF k1[0] OR *no writemask*
    THEN DEST[63:0] :=
            ConvertExpDPFP(SRC2[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[63:0] := 0
        FI
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPSD __m128d _mm_getexp_sd(__m128d a, __m128d b);
VGETEXPSD __m128d _mm_mask_getexp_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VGETEXPSD __m128d _mm_maskz_getexp_sd(__mmask8 k, __m128d a, __m128d b);
VGETEXPSD __m128d _mm_getexp_round_sd(__m128d a, __m128d b, int sae);
VGETEXPSD __m128d _mm_mask_getexp_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int sae);
VGETEXPSD __m128d _mm_maskz_getexp_round_sd(__mmask8 k, __m128d a, __m128d b, int sae);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Table 2-49, "Type E3 Class Exception Conditions."

VGETEXPSH—Convert Exponents of Scalar FP16 Values to FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.MAP6.W0 43 /r VGETEXPSH xmm1{k1}{z}, xmm2, xmm3/m16 {sae}	A	V/V	AVX512_FP16 OR AVX10.1	Convert the exponent of FP16 values in the low word of the source operand to FP16 results representing unbiased integer exponents, and stores the results in the low word of the destination register subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction extracts the biased exponents from the normalized FP16 representation of the low word element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to an unbiased negative integer value. The integer value of the unbiased exponent is converted to an FP16 value and written to the low word element of the destination operand (the first operand) as an FP16 number.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Each GETEXP operation converts the exponent value into a floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-14.

The formula is:

$GETEXP(x) = floor(\log_2(|x|))$

Notation **floor(x)** stands for maximal integer not exceeding real number x.

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTSH). Thus, the VGETEXPSH instruction does not require software to handle SIMD floating-point exceptions.

Operation

VGETEXPSH dest{k1}, src1, src2

```
IF k1[0] or *no writemask*:
    DEST.fp16[0] := getexp_fp16(src2.fp16[0]) // see VGETEXPPH
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
//else DEST.fp16[0] remains unchanged

DEST[127:16] := src1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPSH __m128h _mm_getexp_round_sh (__m128h a, __m128h b, const int sae);
VGETEXPSH __m128h _mm_mask_getexp_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, const int sae);
VGETEXPSH __m128h _mm_maskz_getexp_round_sh (__mmask8 k, __m128h a, __m128h b, const int sae);
VGETEXPSH __m128h _mm_getexp_sh (__m128h a, __m128h b);
VGETEXPSH __m128h _mm_mask_getexp_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VGETEXPSH __m128h _mm_maskz_getexp_sh (__mmask8 k, __m128h a, __m128h b);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

EVEX-encoded instructions, see Table 2-49, "Type E3 Class Exception Conditions."

VGETEXPSS—Convert Exponents of Scalar Single Precision Floating-Point Value to Single Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W0 43 /r VGETEXPSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	A	V/V	AVX512F OR AVX10.1	Convert the biased exponent (bits 30:23) of the low single-precision floating-point value in xmm3/m32 to a single-precision floating-point value representing unbiased integer exponent. Stores the result to xmm1 under the writemask k1 and merge with the other elements of xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Extracts the biased exponent from the normalized single precision floating-point representation of the low double-word data element of the source operand (the third operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. The integer value of the unbiased exponent is converted to single precision floating-point value and written to the destination operand (the first operand) as single precision floating-point numbers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be a XMM register, the source operand can be a XMM register or a float32 memory location.

If writemasking is used, the low doubleword element of the destination operand is conditionally updated depending on the value of writemask register k1. If writemasking is not used, the low doubleword element of the destination operand is unconditionally updated.

Each GETEXP operation converts the exponent value into a floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-15.

The formula is:

GETEXP(x) = floor($\log_2$(|x|))

Notation **floor(x)** stands for maximal integer not exceeding real number x.

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPD). Thus VGETEXPxx instruction do not require software to handle SIMD floating-point exceptions.

Operation

```
// NormalizeExpTinySPFP(SRC[31:0]) is defined in the Operation section of VGETEXPPS
// ConvertExpSPFP(SRC[31:0]) is defined in the Operation section of VGETEXPPS
```

VGETEXPSS (EVEX encoded version)

```
IF k1[0] OR *no writemask*
    THEN DEST[31:0] :=
            ConvertExpDPFP(SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[31:0]:= 0
            FI
    FI;
ENDFOR
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPSS __m128 _mm_getexp_ss(__m128 a, __m128 b);
VGETEXPSS __m128 _mm_mask_getexp_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VGETEXPSS __m128 _mm_maskz_getexp_ss(__mmask8 k, __m128 a, __m128 b);
VGETEXPSS __m128 _mm_getexp_round_ss(__m128 a, __m128 b, int sae);
VGETEXPSS __m128 _mm_mask_getexp_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int sae);
VGETEXPSS __m128 _mm_maskz_getexp_round_ss(__mmask8 k, __m128 a, __m128 b, int sae);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Table 2-49, "Type E3 Class Exception Conditions."

VGETMANTPD—Extract Float64 Vector of Normalized Mantissas From Float64 Vector

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 26 /r ib VGETMANTPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Get Normalized Mantissa from float64 vector xmm2/m128/m64bcst and store the result in xmm1, using imm8 for sign control and mantissa interval normalization, under writemask.
EVEX.256.66.0F3A.W1 26 /r ib VGETMANTPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Get Normalized Mantissa from float64 vector ymm2/m256/m64bcst and store the result in ymm1, using imm8 for sign control and mantissa interval normalization, under writemask.
EVEX.512.66.0F3A.W1 26 /r ib VGETMANTPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, imm8	A	V/V	AVX512F OR AVX10.1	Get Normalized Mantissa from float64 vector zmm2/m512/m64bcst and store the result in zmm1, using imm8 for sign control and mantissa interval normalization, under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Convert double precision floating values in the source operand (the second operand) to double precision floating-point values with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted results are written to the destination operand (the first operand) using writemask k1. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The destination operand is a ZMM/YMM/XMM register updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

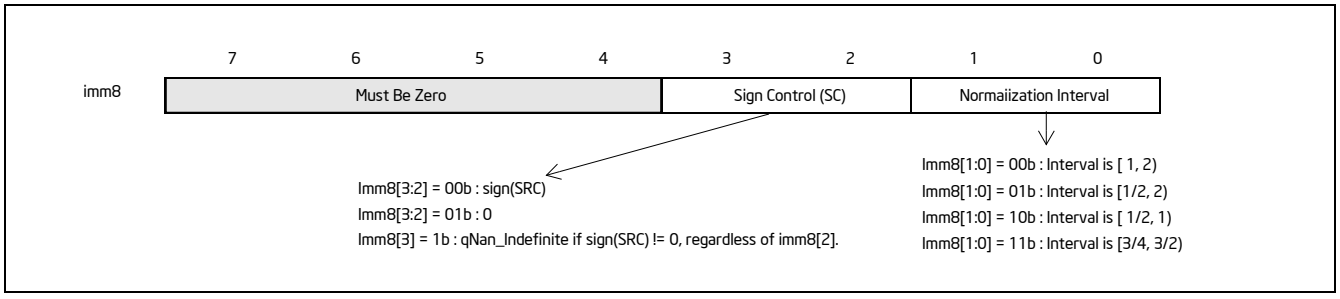


Figure 5-15. Imm8 Controls for VGETMANTPD/SD/PS/SS

For each input double precision floating-point value x, The conversion operation is:

$$GetMant(x) = \pm 2^{k}|x.significand|$$

where:

$$1 <= |x.significand| < 2$$

Unbiased exponent k can be either 0 or -1, depending on the interval range defined by interv, the range of the significand and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign. The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

Each converted double precision floating-point result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-16 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into the destination. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Note: EVEX.vvvv is reserved and must be 1111b; otherwise instructions will #UD.

Table 5-16. GetMant() Special Float Values Behavior

Input	Result	Exceptions / Comments
NaN	QNaN(SRC)	Ignore *interv* If (SRC = SNaN) then #IE
+∞	1.0	Ignore *interv*
+0	1.0	Ignore *interv*
-0	IF (SC[0]) THEN +1.0 ELSE -1.0	Ignore *interv*

-∞	IF (SC[1]) THEN {QNaN_Indefinite}
	ELSE {
	IF (SC[0]) THEN +1.0
	ELSE -1.0
		Ignore *interv*
		If (SC[1]) then #IE

negative	SC[1] ? QNaN_Indefinite : Getmant(SRC)[1]	If (SC[1]) then #IE

NOTES:

1. In case SC[1]==0, the sign of Getmant(SRC) is declared according to SC[0].

Operation

```
def getmant_fp64(src, sign_control, normalization_interval):
    bias := 1023
    dst.sign := sign_control[0] ? 0 : src.sign
    signed_one := sign_control[0] ? +1.0 : -1.0
    dst.exp := src.exp
    dst.fraction := src.fraction
    zero := (dst.exp = 0) and ((dst.fraction = 0) or (MXCSR.DAZ=1))
    denormal := (dst.exp = 0) and (dst.fraction != 0) and (MXCSR.DAZ=0)
    infinity := (dst.exp = 0x7FF) and (dst.fraction = 0)
    nan := (dst.exp = 0x7FF) and (dst.fraction != 0)
    src_signaling := src.fraction[51]
    snan := nan and (src_signaling = 0)
    positive := (src.sign = 0)
    negative := (src.sign = 1)
    if nan:
```

```
        if snan:
            MXCSR.IE := 1
        return qnan(src)

    if positive and (zero or infinity):
        return 1.0
    if negative:
        if zero:
            return signed_one
        if infinity:
            if sign_control[1]:
                MXCSR.IE := 1
                return QNaN_Indefinite
            return signed_one
        if sign_control[1]:
            MXCSR.IE := 1
            return QNaN_Indefinite

    if denormal:
        jbit := 0
        dst.exp := bias
        while jbit = 0:
            jbit := dst.fraction[51]
            dst.fraction := dst.fraction << 1
            dst.exp : = dst.exp - 1
        MXCSR.DE := 1

    unbiased_exp := dst.exp - bias
    odd_exp := unbiased_exp[0]
    signaling_bit := dst.fraction[51]
    if normalization_interval = 0b00:
        dst.exp := bias
    else if normalization_interval = 0b01:
        dst.exp := odd_exp ? bias-1 : bias
    else if normalization_interval = 0b10:
        dst.exp := bias-1
    else if normalization_interval = 0b11:
        dst.exp := signaling_bit ? bias-1 : bias
    return dst
```

VGETMANTPD (EVEX Encoded Versions)

```
VGETMANTPD dest{k1}, src, imm8
VL = 128, 256, or 512
KL := VL / 64
sign_control := imm8[3:2]
normalization_interval := imm8[1:0]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.double[0]
        ELSE:
            tsrc := src.double[i]
        DEST.double[i] := getmant_fp64(tsrc, sign_control, normalization_interval)
    ELSE IF *zeroing*:
        DEST.double[i] := 0
    //else DEST.double[i] remains unchanged

DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VGETMANTPD __m512d _mm512_getmant_pd( __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_mask_getmant_pd(__m512d s, __mmask8 k, __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_maskz_getmant_pd( __mmask8 k, __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_getmant_round_pd( __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m512d _mm512_mask_getmant_round_pd(__m512d s, __mmask8 k, __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m512d _mm512_maskz_getmant_round_pd( __mmask8 k, __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m256d _mm256_getmant_pd( __m256d a, enum intv, enum sgn);
VGETMANTPD __m256d _mm256_mask_getmant_pd(__m256d s, __mmask8 k, __m256d a, enum intv, enum sgn);
VGETMANTPD __m256d _mm256_maskz_getmant_pd( __mmask8 k, __m256d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_getmant_pd( __m128d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_mask_getmant_pd(__m128d s, __mmask8 k, __m128d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_maskz_getmant_pd( __mmask8 k, __m128d a, enum intv, enum sgn);
```

SIMD Floating-Point Exceptions

Denormal, Invalid.

Other Exceptions

See Table 2-48, "Type E2 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VGETMANTPH—Extract FP16 Vector of Normalized Mantissas from FP16 Vector

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.0F3A.W0 26 /r /ib VGETMANTPH xmm1{k1}{z}, xmm2/m128/m16bcst, imm8	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Get normalized mantissa from FP16 vector xmm2/m128/m16bcst and store the result in xmm1, using imm8 for sign control and mantissa interval normalization, subject to writemask k1.
EVEX.256.NP.0F3A.W0 26 /r /ib VGETMANTPH ymm1{k1}{z}, ymm2/m256/m16bcst, imm8	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Get normalized mantissa from FP16 vector ymm2/m256/m16bcst and store the result in ymm1, using imm8 for sign control and mantissa interval normalization, subject to writemask k1.
EVEX.512.NP.0F3A.W0 26 /r /ib VGETMANTPH zmm1{k1}{z}, zmm2/m512/m16bcst {sae}, imm8	A	V/V	AVX512_FP16 OR AVX10.1	Get normalized mantissa from FP16 vector zmm2/m512/m16bcst and store the result in zmm1, using imm8 for sign control and mantissa interval normalization, subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8 (r)	N/A

Description

This instruction converts the FP16 values in the source operand (the second operand) to FP16 values with the mantissa normalization and sign control specified by the imm8 byte, see Table 5-17. The converted results are written to the destination operand (the first operand) using writemask k1. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (SC) is specified by bits 3:2 of the immediate byte.

The destination elements are updated according to the writemask.

Table 5-17. imm8 Controls for VGETMANTPH/VGETMANTSH

imm8 Bits	Definition
imm8[7:4]	Must be zero.
imm8[3:2]	Sign Control (SC) 0b00: Sign(SRC) 0b01: 0 0b1x: QNaN_Indefinite if sign(SRC)!=0
imm8[1:0]	Interv 0b00: Interval is [1, 2) 0b01: Interval is [1/2, 2) 0b10: Interval is [1/2, 1) 0b11: Interval is [3/4, 3/2)

For each input FP16 value x, The conversion operation is:

$$GetMant(x) = \pm 2^{k}|x.significand|$$

where:

$$1 \leq |x.significand| < 2$$

Unbiased exponent k depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by the sign control and the source sign and the leading fraction bit.

The encoded value of imm8[1:0] and sign control are shown in Table 5-17.

Each converted FP16 result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-18 when dealing with floating-point special numbers.

Table 5-18. GetMant() Special Float Values Behavior

Input	Result	Exceptions / Comments
NaN	QNaN(SRC)	Ignore *interv.* If (SRC = SNaN), then #IE.
+∞	1.0	Ignore *interv.*
+0	1.0	Ignore *interv.*
-0	IF (SC[0]) THEN +1.0 ELSE -1.0	Ignore *interv.*

-∞	IF (SC[1]) THEN {QNaN_Indefinite}
	ELSE {
	IF (SC[0]) THEN +1.0
	ELSE -1.0
		Ignore *interv.*
		If (SC[1]), then #IE.

negative	SC[1] ? QNaN_Indefinite : Getmant(SRC)[1]	If (SC[1]), then #IE.

NOTES:

1. In case SC[1]==0, the sign of Getmant(SRC) is declared according to SC[0].

Operation

```
def getmant_fp16(src, sign_control, normalization_interval):
    bias := 15
    dst.sign := sign_control[0] ? 0 : src.sign
    signed_one := sign_control[0] ? +1.0 : -1.0
    dst.exp := src.exp
    dst.fraction := src.fraction
    zero := (dst.exp = 0) and (dst.fraction = 0)
    denormal := (dst.exp = 0) and (dst.fraction != 0)
    infinity := (dst.exp = 0x1F) and (dst.fraction = 0)
    nan := (dst.exp = 0x1F) and (dst.fraction != 0)
    src_signaling := src.fraction[9]
    snan := nan and (src_signaling = 0)
    positive := (src.sign = 0)
    negative := (src.sign = 1)
    if nan:
        if snan:
            MXCSR.IE := 1
        return qnan(src)

    if positive and (zero or infinity):
        return 1.0
    if negative:
        if zero:
            return signed_one
        if infinity:
```

```
                if sign_control[1]:
                    MXCSR.IE := 1
                    return QNaN_Indefinite
                return signed_one
            if sign_control[1]:
                MXCSR.IE := 1
                return QNaN_Indefinite
        if denormal:
            jbit := 0
            dst.exp := bias             // set exponent to bias value
            while jbit = 0:
                jbit := dst.fraction[9]
                dst.fraction := dst.fraction << 1
                dst.exp : = dst.exp - 1
            MXCSR.DE := 1

        unbaiased_exp := dst.exp - bias
        odd_exp := unbaiased_exp[0]
        signaling_bit := dst.fraction[9]
        if normalization_interval = 0b00:
            dst.exp := bias
        else if normalization_interval = 0b01:
            dst.exp := odd_exp ? bias-1 : bias
        else if normalization_interval = 0b10:
            dst.exp := bias-1
        else if normalization_interval = 0b11:
            dst.exp := signaling_bit ? bias-1 : bias
        return dst
```

VGETMANTPH dest{k1}, src, imm8

```
VL = 128, 256 or 512
KL := VL/16

sign_control := imm8[3:2]
normalization_interval := imm8[1:0]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.fp16[0]
        ELSE:
            tsrc := src.fp16[i]
        DEST.fp16[i] := getmant_fp16(tsrc, sign_control, normalization_interval)
    ELSE IF *zeroing*:
        DEST.fp16[i] := 0
    //else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGETMANTPH __m128h _mm_getmant_ph (__m128h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m128h _mm_mask_getmant_ph (__m128h src, __mmask8 k, __m128h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m128h _mm_maskz_getmant_ph (__mmask8 k, __m128h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m256h _mm256_getmant_ph (__m256h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m256h _mm256_mask_getmant_ph (__m256h src, __mmask16 k, __m256h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m256h _mm256_maskz_getmant_ph (__mmask16 k, __m256h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m512h _mm512_getmant_ph (__m512h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m512h _mm512_mask_getmant_ph (__m512h src, __mmask32 k, __m512h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m512h _mm512_maskz_getmant_ph (__mmask32 k, __m512h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m512h _mm512_getmant_round_ph (__m512h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign, const int sae);
VGETMANTPH __m512h _mm512_mask_getmant_round_ph (__m512h src, __mmask32 k, __m512h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign, const int sae);
VGETMANTPH __m512h _mm512_maskz_getmant_round_ph (__mmask32 k, __m512h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign, const int sae);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VGETMANTPS—Extract Float32 Vector of Normalized Mantissas From Float32 Vector

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 26 /r ib VGETMANTPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Get normalized mantissa from float32 vector xmm2/m128/m32bcst and store the result in xmm1, using imm8 for sign control and mantissa interval normalization, under writemask.
EVEX.256.66.0F3A.W0 26 /r ib VGETMANTPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Get normalized mantissa from float32 vector ymm2/m256/m32bcst and store the result in ymm1, using imm8 for sign control and mantissa interval normalization, under writemask.
EVEX.512.66.0F3A.W0 26 /r ib VGETMANTPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}, imm8	A	V/V	AVX512F OR AVX10.1	Get normalized mantissa from float32 vector zmm2/m512/m32bcst and store the result in zmm1, using imm8 for sign control and mantissa interval normalization, under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Convert single precision floating values in the source operand (the second operand) to single precision floating-point values with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted results are written to the destination operand (the first operand) using writemask k1. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The destination operand is a ZMM/YMM/XMM register updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

For each input single precision floating-point value x, The conversion operation is:

$$GetMant(x) = \pm 2^{k}|x.significand|$$

where:

$$1 <= |x.significand| < 2$$

Unbiased exponent k can be either 0 or -1, depending on the interval range defined by interv, the range of the significand and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign. The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

Each converted single precision floating-point result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-16 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into the destination. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Note: EVEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

```
def getmant_fp32(src, sign_control, normalization_interval):
    bias := 127
    dst.sign := sign_control[0] ? 0 : src.sign
    signed_one := sign_control[0] ? +1.0 : -1.0
    dst.exp := src.exp
    dst.fraction := src.fraction
    zero := (dst.exp = 0) and ((dst.fraction = 0) or (MXCSR.DAZ=1))
    denormal := (dst.exp = 0) and (dst.fraction != 0) and (MXCSR.DAZ=0)
    infinity := (dst.exp = 0xFF) and (dst.fraction = 0)
    nan := (dst.exp = 0xFF) and (dst.fraction != 0)
    src_signaling := src.fraction[22]
    snan := nan and (src_signaling = 0)
    positive := (src.sign = 0)
    negative := (src.sign = 1)
    if nan:
        if snan:
            MXCSR.IE := 1
        return qnan(src)

    if positive and (zero or infinity):
        return 1.0
    if negative:
        if zero:
            return signed_one
        if infinity:
            if sign_control[1]:
                MXCSR.IE := 1
                return QNaN_Indefinite
            return signed_one
        if sign_control[1]:
            MXCSR.IE := 1
            return QNaN_Indefinite

    if denormal:
        jbit := 0
        dst.exp := bias
        while jbit = 0:
            jbit := dst.fraction[22]
            dst.fraction := dst.fraction << 1
            dst.exp : = dst.exp - 1
        MXCSR.DE := 1

    unbiased_exp := dst.exp - bias
    odd_exp  := unbiased_exp[0]
    signaling_bit := dst.fraction[22]
    if normalization_interval = 0b00:
        dst.exp := bias
    else if normalization_interval = 0b01:
        dst.exp := odd_exp ? bias-1 : bias
    else if normalization_interval = 0b10:
        dst.exp := bias-1
    else if normalization_interval = 0b11:
        dst.exp := signaling_bit ? bias-1 : bias
```

```
    return dst
```

VGETMANTPS (EVEX encoded versions)

```
VGETMANTPS dest{k1}, src, imm8
VL = 128, 256, or 512
KL := VL / 32
sign_control := imm8[3:2]
normalization_interval := imm8[1:0]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.float[0]
        ELSE:
            tsrc := src.float[i]
        DEST.float[i] := getmant_fp32(tsrc, sign_control, normalization_interval)
    ELSE IF *zeroing*:
        DEST.float[i] := 0
    //else DEST.float[i] remains unchanged

DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VGETMANTPS __m512 _mm512_getmant_ps( __m512 a, enum intv, enum sgn);
VGETMANTPS __m512 _mm512_mask_getmant_ps(__m512 s, __mmask16 k, __m512 a, enum intv, enum sgn;
VGETMANTPS __m512 _mm512_maskz_getmant_ps(__mmask16 k, __m512 a, enum intv, enum sgn);
VGETMANTPS __m512 _mm512_getmant_round_ps( __m512 a, enum intv, enum sgn, int r);
VGETMANTPS __m512 _mm512_mask_getmant_round_ps(__m512 s, __mmask16 k, __m512 a, enum intv, enum sgn, int r);
VGETMANTPS __m512 _mm512_maskz_getmant_round_ps(__mmask16 k, __m512 a, enum intv, enum sgn, int r);
VGETMANTPS __m256 _mm256_getmant_ps( __m256 a, enum intv, enum sgn);
VGETMANTPS __m256 _mm256_mask_getmant_ps(__m256 s, __mmask8 k, __m256 a, enum intv, enum sgn);
VGETMANTPS __m256 _mm256_maskz_getmant_ps( __mmask8 k, __m256 a, enum intv, enum sgn);
VGETMANTPS __m128 _mm_getmant_ps( __m128 a, enum intv, enum sgn);
VGETMANTPS __m128 _mm_mask_getmant_ps(__m128 s, __mmask8 k, __m128 a, enum intv, enum sgn);
VGETMANTPS __m128 _mm_maskz_getmant_ps( __mmask8 k, __m128 a, enum intv, enum sgn);
```

SIMD Floating-Point Exceptions

Denormal, Invalid.

Other Exceptions

See Table 2-48, "Type E2 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VGETMANTSD—Extract Float64 of Normalized Mantissa From Float64 Scalar

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W1 27 /r ib VGETMANTSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	A	V/V	AVX512F OR AVX10.1	Extract the normalized mantissa of the low float64 element in xmm3/m64 using imm8 for sign control and mantissa interval normalization. Store the mantissa to xmm1 under the writemask k1 and merge with the other elements of xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Convert the double precision floating values in the low quadword element of the second source operand (the third operand) to double precision floating-point value with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted result is written to the low quadword element of the destination operand (the first operand) using writemask k1. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The conversion operation is:

$$GetMant(x) = \pm 2^{k}|x.significand|$$

where:

$$1 <= |x.significand| < 2$$

Unbiased exponent k can be either 0 or -1, depending on the interval range defined by interv, the range of the significand and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign. The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

The converted double precision floating-point result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-16 when dealing with floating-point special numbers.

If writemasking is used, the low quadword element of the destination operand is conditionally updated depending on the value of writemask register k1. If writemasking is not used, the low quadword element of the destination operand is unconditionally updated.

Operation

```
// getmant_fp64(src, sign_control, normalization_interval) is defined in the operation section of VGETMANTPD

VGETMANTSD (EVEX encoded version)
SignCtrl[1:0] := IMM8[3:2];
Interv[1:0] := IMM8[1:0];
IF k1[0] OR *no writemask*
    THEN DEST[63:0] :=
            getmant_fp64(src, sign_control, normalization_interval)
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63:0] := 0
        FI
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VGETMANTSD __m128d _mm_getmant_sd( __m128d a, __m128 b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_mask_getmant_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_maskz_getmant_sd( __mmask8 k, __m128 a, __m128d b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_getmant_round_sd( __m128d a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSD __m128d _mm_mask_getmant_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);
VGETMANTSD __m128d _mm_maskz_getmant_round_sd( __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);
```

SIMD Floating-Point Exceptions

Denormal, Invalid

Other Exceptions

See Table 2-49, "Type E3 Class Exception Conditions."

VGETMANTSH—Extract FP16 of Normalized Mantissa from FP16 Scalar

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.NP.0F3A.W0 27 /r /ib VGETMANTSH xmm1{k1}{z}, xmm2, xmm3/m16 {sae}, imm8	A	V/V	AVX512_FP16 OR AVX10.1	Extract the normalized mantissa of the low FP16 element in xmm3/m16 using imm8 for sign control and mantissa interval normalization. Store the mantissa to xmm1 subject to writemask k1 and merge with the other elements of xmm2. Bits 127:16 of xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

This instruction converts the FP16 value in the low element of the second source operand to FP16 values with the mantissa normalization and sign control specified by the imm8 byte, see Table 5-17. The converted result is written to the low element of the destination operand using writemask k1. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (SC) is specified by bits 3:2 of the immediate byte.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

For each input FP16 value x, The conversion operation is:

$$\text{GetMant}(x) = \pm 2^{k}|x.\text{significand}|$$

where:

$$1 \leq |x.\text{significand}| < 2$$

Unbiased exponent k depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by the sign control and the source sign and the leading fraction bit.

The encoded value of imm8[1:0] and sign control are shown in Table 5-17.

Each converted FP16 result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-18 when dealing with floating-point special numbers.

Operation

VGETMANTSH dest{k1}, src1, src2, imm8

```
sign_control := imm8[3:2]
normalization_interval := imm8[1:0]

IF k1[0] or *no writemask*:
    dest.fp16[0] := getmant_fp16(src2.fp16[0],     // see VGETMANTPH
                                 sign_control,
                                 normalization_interval)
ELSE IF *zeroing*:
    dest.fp16[0] := 0
//else dest.fp16[0] remains unchanged

DEST[127:16] := src1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGETMANTSH __m128h _mm_getmant_round_sh (__m128h a, __m128h b, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign, const int sae);

VGETMANTSH __m128h _mm_mask_getmant_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign, const int sae);

VGETMANTSH __m128h _mm_maskz_getmant_round_sh (__mmask8 k, __m128h a, __m128h b, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign, const int sae);

VGETMANTSH __m128h _mm_getmant_sh (__m128h a, __m128h b, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);

VGETMANTSH __m128h _mm_mask_getmant_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);

VGETMANTSH __m128h _mm_maskz_getmant_sh (__mmask8 k, __m128h a, __m128h b, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VGETMANTSS—Extract Float32 Vector of Normalized Mantissa From Float32 Scalar

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W0 27 /r ib VGETMANTSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512F OR AVX10.1	Extract the normalized mantissa from the low float32 element of xmm3/m32 using imm8 for sign control and mantissa interval normalization, store the mantissa to xmm1 under the writemask k1 and merge with the other elements of xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Convert the single precision floating values in the low doubleword element of the second source operand (the third operand) to single precision floating-point value with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted result is written to the low doubleword element of the destination operand (the first operand) using writemask k1. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The conversion operation is:

$$GetMant(x) = \pm 2^{k}|x.significand|$$

where:

$$1 <= |x.significand| < 2$$

Unbiased exponent k can be either 0 or -1, depending on the interval range defined by interv, the range of the significand and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign. The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

The converted single precision floating-point result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-16 when dealing with floating-point special numbers.

If writemasking is used, the low doubleword element of the destination operand is conditionally updated depending on the value of writemask register k1. If writemasking is not used, the low doubleword element of the destination operand is unconditionally updated.

Operation

```
// getmant_fp32(src, sign_control, normalization_interval) is defined in the operation section of VGETMANTPS

VGETMANTSS (EVEX encoded version)
SignCtrl[1:0] := IMM8[3:2];
Interv[1:0] := IMM8[1:0];
IF k1[0] OR *no writemask*
    THEN DEST[31:0] :=
            getmant_fp32(src, sign_control, normalization_interval)
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] := 0
        FI
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VGETMANTSS __m128 _mm_getmant_ss( __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_mask_getmant_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_maskz_getmant_ss( __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_getmant_round_ss( __m128 a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSS __m128 _mm_mask_getmant_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSS __m128 _mm_maskz_getmant_round_ss( __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn, int r);
```

SIMD Floating-Point Exceptions

Denormal, Invalid

Other Exceptions

See Table 2-49, "Type E3 Class Exception Conditions."

VINSERTF128/VINSERTF32x4/VINSERTF64x2/VINSERTF32x8/VINSERTF64x4—Insert Packed Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 18 /r ib VINSERTF128 ymm1, ymm2, xmm3/m128, imm8	A	V/V	AVX	Insert 128 bits of packed floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1.
EVEX.256.66.0F3A.W0 18 /r ib VINSERTF32X4 ymm1 {k1}{z}, ymm2, xmm3/m128, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Insert 128 bits of packed single-precision floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.
EVEX.512.66.0F3A.W0 18 /r ib VINSERTF32X4 zmm1 {k1}{z}, zmm2, xmm3/m128, imm8	C	V/V	AVX512F OR AVX10.1	Insert 128 bits of packed single-precision floating-point values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.256.66.0F3A.W1 18 /r ib VINSERTF64X2 ymm1 {k1}{z}, ymm2, xmm3/m128, imm8	B	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Insert 128 bits of packed double precision floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.
EVEX.512.66.0F3A.W1 18 /r ib VINSERTF64X2 zmm1 {k1}{z}, zmm2, xmm3/m128, imm8	B	V/V	AVX512DQ OR AVX10.1	Insert 128 bits of packed double precision floating-point values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.512.66.0F3A.W0 1A /r ib VINSERTF32X8 zmm1 {k1}{z}, zmm2, ymm3/m256, imm8	D	V/V	AVX512DQ OR AVX10.1	Insert 256 bits of packed single-precision floating-point values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.512.66.0F3A.W1 1A /r ib VINSERTF64X4 zmm1 {k1}{z}, zmm2, ymm3/m256, imm8	C	V/V	AVX512F OR AVX10.1	Insert 256 bits of packed double precision floating-point values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
B	Tuple2	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple4	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8
D	Tuple8	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

VINSERTF128/VINSERTF32x4 and VINSERTF64x2 insert 128-bits of packed floating-point values from the second source operand (the third operand) into the destination operand (the first operand) at an 128-bit granularity offset multiplied by imm8[0] (256-bit) or imm8[1:0]. The remaining portions of the destination operand are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The destination and first source operands are vector registers.

VINSERTF32x4: The destination operand is a ZMM/YMM register and updated at 32-bit granularity according to the writemask. The high 6/7 bits of the immediate are ignored.

VINSERTF64x2: The destination operand is a ZMM/YMM register and updated at 64-bit granularity according to the writemask. The high 6/7 bits of the immediate are ignored.

VINSERTF32x8 and VINSERTF64x4 inserts 256-bits of packed floating-point values from the second source operand (the third operand) into the destination operand (the first operand) at a 256-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an YMM register or a 256-bit memory location. The high 7 bits of the immediate are ignored. The destination operand is a ZMM register and updated at 32/64-bit granularity according to the writemask.

Operation

VINSERTF32x4 (EVEX encoded versions)

```
(KL, VL) = (8, 256), (16, 512)
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
IF VL = 256
    CASE (imm8[0]) OF
        0:    TMP_DEST[127:0] := SRC2[127:0]
        1:    TMP_DEST[255:128] := SRC2[127:0]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00:  TMP_DEST[127:0] := SRC2[127:0]
        01:  TMP_DEST[255:128] := SRC2[127:0]
        10:  TMP_DEST[383:256] := SRC2[127:0]
        11:  TMP_DEST[511:384] := SRC2[127:0]
    ESAC.
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VINSERTF64x2 (EVEX encoded versions)

```
(KL, VL) = (4, 256), (8, 512)
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
IF VL = 256
    CASE (imm8[0]) OF
        0:    TMP_DEST[127:0] := SRC2[127:0]
        1:    TMP_DEST[255:128] := SRC2[127:0]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00:  TMP_DEST[127:0] := SRC2[127:0]
        01:  TMP_DEST[255:128] := SRC2[127:0]
        10:  TMP_DEST[383:256] := SRC2[127:0]
        11:  TMP_DEST[511:384] := SRC2[127:0]
    ESAC.
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VINSERTF32x8 (EVEX.U1.512 encoded version)

```
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC2[255:0]
    1: TMP_DEST[511:256] := SRC2[255:0]
ESAC.

FOR j := 0 TO 15
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VINSERTF64x4 (EVEX.512 encoded version)

```
VL = 512
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC2[255:0]
    1: TMP_DEST[511:256] := SRC2[255:0]
ESAC.

FOR j := 0 TO 7
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VINSERTF128 (VEX encoded version)

```
TEMP[255:0] := SRC1[255:0]
CASE (imm8[0]) OF
    0: TEMP[127:0] := SRC2[127:0]
    1: TEMP[255:128] := SRC2[127:0]
ESAC
DEST := TEMP
```

Intel C/C++ Compiler Intrinsic Equivalent

VINSERTF32x4 __m512 _mm512_insertf32x4(__m512 a, __m128 b, int imm);
VINSERTF32x4 __m512 _mm512_mask_insertf32x4(__m512 s, __mmask16 k, __m512 a, __m128 b, int imm);
VINSERTF32x4 __m512 _mm512_maskz_insertf32x4(__mmask16 k, __m512 a, __m128 b, int imm);
VINSERTF32x4 __m256 _mm256_insertf32x4(__m256 a, __m128 b, int imm);
VINSERTF32x4 __m256 _mm256_mask_insertf32x4(__m256 s, __mmask8 k, __m256 a, __m128 b, int imm);
VINSERTF32x4 __m256 _mm256_maskz_insertf32x4(__mmask8 k, __m256 a, __m128 b, int imm);
VINSERTF32x8 __m512 _mm512_insertf32x8(__m512 a, __m256 b, int imm);
VINSERTF32x8 __m512 _mm512_mask_insertf32x8(__m512 s, __mmask16 k, __m512 a, __m256 b, int imm);
VINSERTF32x8 __m512 _mm512_maskz_insertf32x8(__mmask16 k, __m512 a, __m256 b, int imm);
VINSERTF64x2 __m512d _mm512_insertf64x2(__m512d a, __m128d b, int imm);
VINSERTF64x2 __m512d _mm512_mask_insertf64x2(__m512d s, __mmask8 k, __m512d a, __m128d b, int imm);
VINSERTF64x2 __m512d _mm512_maskz_insertf64x2(__mmask8 k, __m512d a, __m128d b, int imm);
VINSERTF64x2 __m256d _mm256_insertf64x2(__m256d a, __m128d b, int imm);
VINSERTF64x2 __m256d _mm256_mask_insertf64x2(__m256d s, __mmask8 k, __m256d a, __m128d b, int imm);
VINSERTF64x2 __m256d _mm256_maskz_insertf64x2(__mmask8 k, __m256d a, __m128d b, int imm);
VINSERTF64x4 __m512d _mm512_insertf64x4(__m512d a, __m256d b, int imm);
VINSERTF64x4 __m512d _mm512_mask_insertf64x4(__m512d s, __mmask8 k, __m512d a, __m256d b, int imm);
VINSERTF64x4 __m512d _mm512_maskz_insertf64x4(__mmask8 k, __m512d a, __m256d b, int imm);
VINSERTF128 __m256 _mm256_insertf128_ps (__m256 a, __m128 b, int offset);
VINSERTF128 __m256d _mm256_insertf128_pd (__m256d a, __m128d b, int offset);
VINSERTF128 __m256i _mm256_insertf128_si256 (__m256i a, __m128i b, int offset);

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instruction, see Table 2-23, "Type 6 Class Exception Conditions."
Additionally:

#UD	If VEX.L = 0.

EVEX-encoded instruction, see Table 2-56, "Type E6NF Class Exception Conditions."

VINSERTI128/VINSERTI32x4/VINSERTI64x2/VINSERTI32x8/VINSERTI64x4—Insert Packed Integer Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.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/m128 and the remaining values from ymm2 into ymm1.
EVEX.256.66.0F3A.W0 38 /r ib VINSERTI32X4 ymm1 {k1}{z}, ymm2, xmm3/m128, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Insert 128 bits of packed doubleword integer values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.
EVEX.512.66.0F3A.W0 38 /r ib VINSERTI32X4 zmm1 {k1}{z}, zmm2, xmm3/m128, imm8	C	V/V	AVX512F OR AVX10.1	Insert 128 bits of packed doubleword integer values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.256.66.0F3A.W1 38 /r ib VINSERTI64X2 ymm1 {k1}{z}, ymm2, xmm3/m128, imm8	B	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Insert 128 bits of packed quadword integer values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.
EVEX.512.66.0F3A.W1 38 /r ib VINSERTI64X2 zmm1 {k1}{z}, zmm2, xmm3/m128, imm8	B	V/V	AVX512DQ OR AVX10.1	Insert 128 bits of packed quadword integer values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.512.66.0F3A.W0 3A /r ib VINSERTI32X8 zmm1 {k1}{z}, zmm2, ymm3/m256, imm8	D	V/V	AVX512DQ OR AVX10.1	Insert 256 bits of packed doubleword integer values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.512.66.0F3A.W1 3A /r ib VINSERTI64X4 zmm1 {k1}{z}, zmm2, ymm3/m256, imm8	C	V/V	AVX512F OR AVX10.1	Insert 256 bits of packed quadword integer values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
B	Tuple2	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple4	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8
D	Tuple8	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

VINSERTI32x4 and VINSERTI64x2 inserts 128-bits of packed integer values from the second source operand (the third operand) into the destination operand (the first operand) at an 128-bit granular offset multiplied by imm8[0] (256-bit) or imm8[1:0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The high 6/7bits of the immediate are ignored. The destination operand is a ZMM/YMM register and updated at 32 and 64-bit granularity according to the writemask.

VINSERTI32x8 and VINSERTI64x4 inserts 256-bits of packed integer values from the second source operand (the third operand) into the destination operand (the first operand) at a 256-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an YMM register or a 256-bit memory location. The

upper bits of the immediate are ignored. The destination operand is a ZMM register and updated at 32 and 64-bit granularity according to the writemask.

VINSERTI128 inserts 128-bits of packed integer data from the second source operand (the third operand) into the destination operand (the first operand) at a 128-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The high 7 bits of the immediate are ignored. VEX.L must be 1, otherwise attempt to execute this instruction with VEX.L=0 will cause #UD.

Operation

VINSERTI32x4 (EVEX encoded versions)

```
(KL, VL) = (8, 256), (16, 512)
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
IF VL = 256
    CASE (imm8[0]) OF
        0:    TMP_DEST[127:0] := SRC2[127:0]
        1:    TMP_DEST[255:128] := SRC2[127:0]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00:  TMP_DEST[127:0] := SRC2[127:0]
        01:  TMP_DEST[255:128] := SRC2[127:0]
        10:  TMP_DEST[383:256] := SRC2[127:0]
        11:  TMP_DEST[511:384] := SRC2[127:0]
    ESAC.
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VINSERTI64x2 (EVEX encoded versions)

```
(KL, VL) = (4, 256), (8, 512)
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
IF VL = 256
    CASE (imm8[0]) OF
        0:    TMP_DEST[127:0] := SRC2[127:0]
        1:    TMP_DEST[255:128] := SRC2[127:0]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00:  TMP_DEST[127:0] := SRC2[127:0]
        01:  TMP_DEST[255:128] := SRC2[127:0]
        10:  TMP_DEST[383:256] := SRC2[127:0]
        11:  TMP_DEST[511:384] := SRC2[127:0]
    ESAC.
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VINSERTI32x8 (EVEX.U1.512 encoded version)

```
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC2[255:0]
    1: TMP_DEST[511:256] := SRC2[255:0]
ESAC.

FOR j := 0 TO 15
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VINSERTI64x4 (EVEX.512 encoded version)

```
VL = 512
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC2[255:0]
    1: TMP_DEST[511:256] := SRC2[255:0]
ESAC.

FOR j := 0 TO 7
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VINSERTI128

```
TEMP[255:0] := SRC1[255:0]
CASE (imm8[0]) OF
    0: TEMP[127:0] := SRC2[127:0]
    1: TEMP[255:128] := SRC2[127:0]
ESAC
DEST := TEMP
```

Intel C/C++ Compiler Intrinsic Equivalent

VINSERTI32x4 _mm512i _inserti32x4(__m512i a, __m128i b, int imm);
VINSERTI32x4 _mm512i _mask_inserti32x4(__m512i s, __mmask16 k, __m512i a, __m128i b, int imm);
VINSERTI32x4 _mm512i _maskz_inserti32x4(__mmask16 k, __m512i a, __m128i b, int imm);
VINSERTI32x4 __m256i _mm256_inserti32x4(__m256i a, __m128i b, int imm);
VINSERTI32x4 __m256i _mm256_mask_inserti32x4(__m256i s, __mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI32x4 __m256i _mm256_maskz_inserti32x4(__mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI32x8 __m512i _mm512_inserti32x8(__m512i a, __m256i b, int imm);
VINSERTI32x8 __m512i _mm512_mask_inserti32x8(__m512i s, __mmask16 k, __m512i a, __m256i b, int imm);
VINSERTI32x8 __m512i _mm512_maskz_inserti32x8(__mmask16 k, __m512i a, __m256i b, int imm);
VINSERTI64x2 __m512i _mm512_inserti64x2(__m512i a, __m128i b, int imm);
VINSERTI64x2 __m512i _mm512_mask_inserti64x2(__m512i s, __mmask8 k, __m512i a, __m128i b, int imm);
VINSERTI64x2 __m512i _mm512_maskz_inserti64x2(__mmask8 k, __m512i a, __m128i b, int imm);
VINSERTI64x2 __m256i _mm256_inserti64x2(__m256i a, __m128i b, int imm);
VINSERTI64x2 __m256i _mm256_mask_inserti64x2(__m256i s, __mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI64x2 __m256i _mm256_maskz_inserti64x2(__mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI64x4 _mm512_inserti64x4(__m512i a, __m256i b, int imm);
VINSERTI64x4 _mm512_mask_inserti64x4(__m512i s, __mmask8 k, __m512i a, __m256i b, int imm);
VINSERTI64x4 _mm512_maskz_inserti64x4(__mmask m, __m512i a, __m256i b, int imm);
VINSERTI128 __m256i _mm256_insertf128_si256 (__m256i a, __m128i b, int offset);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instruction, see Table 2-23, “Type 6 Class Exception Conditions.”
Additionally:

#UD	If VEX.L = 0.

EVEX-encoded instruction, see Table 2-56, “Type E6NF Class Exception Conditions.”

VMASKMOV—Conditional SIMD Packed Loads and Stores

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 2C /r VMASKMOVPS xmm1, xmm2, m128	RVM	V/V	AVX	Conditionally load packed single precision values from m128 using mask in xmm2 and store in xmm1.
VEX.256.66.0F38.W0 2C /r VMASKMOVPS ymm1, ymm2, m256	RVM	V/V	AVX	Conditionally load packed single precision values from m256 using mask in ymm2 and store in ymm1.
VEX.128.66.0F38.W0 2D /r VMASKMOVPD xmm1, xmm2, m128	RVM	V/V	AVX	Conditionally load packed double precision values from m128 using mask in xmm2 and store in xmm1.
VEX.256.66.0F38.W0 2D /r VMASKMOVPD ymm1, ymm2, m256	RVM	V/V	AVX	Conditionally load packed double precision values from m256 using mask in ymm2 and store in ymm1.
VEX.128.66.0F38.W0 2E /r VMASKMOVPS m128, xmm1, xmm2	MVR	V/V	AVX	Conditionally store packed single precision values from xmm2 using mask in xmm1.
VEX.256.66.0F38.W0 2E /r VMASKMOVPS m256, ymm1, ymm2	MVR	V/V	AVX	Conditionally store packed single precision values from ymm2 using mask in ymm1.
VEX.128.66.0F38.W0 2F /r VMASKMOVPD m128, xmm1, xmm2	MVR	V/V	AVX	Conditionally store packed double precision values from xmm2 using mask in xmm1.
VEX.256.66.0F38.W0 2F /r VMASKMOVPD m256, ymm1, ymm2	MVR	V/V	AVX	Conditionally store packed double precision values from ymm2 using mask in ymm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
MVR	ModRM:r/m (w)	VEX.vvvv (r)	ModRM:reg (r)	N/A

Description

Conditionally moves packed data elements from the second source operand into the corresponding data element of the destination operand, depending on the mask bits associated with each data element. The mask bits are specified in the first source operand.

The mask bit for each data element is the most significant bit of that element in the first source operand. If a mask is 1, the corresponding data element is copied from the second source operand to the destination operand. If the mask is 0, the corresponding data element is set to zero in the load form of these instructions, and unmodified in the store form.

The second source operand is a memory address for the load form of these instruction. The destination operand is a memory address for the store form of these instructions. The other operands are both XMM registers (for VEX.128 version) or YMM registers (for VEX.256 version).

Faults occur only due to mask-bit required memory accesses that caused the faults. Faults will not occur due to referencing any memory location if the corresponding mask bit for that memory location is 0. For example, no faults will be detected if the mask bits are all zero.

Unlike previous MASKMOV instructions (MASKMOVQ and MASKMOVDQU), a nontemporal hint is not applied to these instructions.

Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s.

VMASKMOV should not be used to access memory mapped I/O and un-cached memory as the access and the ordering of the individual loads or stores it does is implementation specific.

In cases where mask bits indicate data should not be loaded or stored paging A and D bits will be set in an implementation dependent way. However, A and D bits are always set for pages where data is actually loaded/stored.

Note: for load forms, the first source (the mask) is encoded in VEX.vvvv; the second source is encoded in rm_field, and the destination register is encoded in reg_field.

Note: for store forms, the first source (the mask) is encoded in VEX.vvvv; the second source register is encoded in reg_field, and the destination memory location is encoded in rm_field.

Operation

VMASKMOVPS -128-bit load

```
DEST[31:0] := IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] := IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] := IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:97] := IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[MAXVL-1:128] := 0
```

VMASKMOVPS - 256-bit load

```
DEST[31:0] := IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] := IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] := IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] := IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[159:128] := IF (SRC1[159]) Load_32(mem + 16) ELSE 0
DEST[191:160] := IF (SRC1[191]) Load_32(mem + 20) ELSE 0
DEST[223:192] := IF (SRC1[223]) Load_32(mem + 24) ELSE 0
DEST[255:224] := IF (SRC1[255]) Load_32(mem + 28) ELSE 0
```

VMASKMOVPD - 128-bit load

```
DEST[63:0] := IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] := IF (SRC1[127]) Load_64(mem + 16) ELSE 0
DEST[MAXVL-1:128] := 0
```

VMASKMOVPD - 256-bit load

```
DEST[63:0] := IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] := IF (SRC1[127]) Load_64(mem + 8) ELSE 0
DEST[195:128] := IF (SRC1[191]) Load_64(mem + 16) ELSE 0
DEST[255:196] := IF (SRC1[255]) Load_64(mem + 24) ELSE 0
```

VMASKMOVPS - 128-bit store

```
IF (SRC1[31]) DEST[31:0] := SRC2[31:0]
IF (SRC1[63]) DEST[63:32] := SRC2[63:32]
IF (SRC1[95]) DEST[95:64] := SRC2[95:64]
IF (SRC1[127]) DEST[127:96] := SRC2[127:96]
```

VMASKMOVPS - 256-bit store

```
IF (SRC1[31]) DEST[31:0] := SRC2[31:0]
IF (SRC1[63]) DEST[63:32] := SRC2[63:32]
IF (SRC1[95]) DEST[95:64] := SRC2[95:64]
IF (SRC1[127]) DEST[127:96] := SRC2[127:96]
IF (SRC1[159]) DEST[159:128] :=SRC2[159:128]
IF (SRC1[191]) DEST[191:160] := SRC2[191:160]
IF (SRC1[223]) DEST[223:192] := SRC2[223:192]
IF (SRC1[255]) DEST[255:224] := SRC2[255:224]
```

VMASKMOVPD - 128-bit store

```
IF (SRC1[63]) DEST[63:0] := SRC2[63:0]
IF (SRC1[127]) DEST[127:64] := SRC2[127:64]
```

VMASKMOVPD - 256-bit store

```
IF (SRC1[63]) DEST[63:0] := SRC2[63:0]
IF (SRC1[127]) DEST[127:64] := SRC2[127:64]
IF (SRC1[191]) DEST[191:128] := SRC2[191:128]
IF (SRC1[255]) DEST[255:192] := SRC2[255:192]
```

Intel C/C++ Compiler Intrinsic Equivalent

__m256 _mm256_maskload_ps(float const *a, __m256i mask)

void _mm256_maskstore_ps(float *a, __m256i mask, __m256 b)

__m256d _mm256_maskload_pd(double *a, __m256i mask);

void _mm256_maskstore_pd(double *a, __m256i mask, __m256d b);

__m128 _mm_maskload_ps(float const *a, __m128i mask)

void _mm_maskstore_ps(float *a, __m128i mask, __m128 b)

__m128d _mm_maskload_pd(double const *a, __m128i mask);

void _mm_maskstore_pd(double *a, __m128i mask, __m128d b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-23, "Type 6 Class Exception Conditions" (No AC# reported for any mask bit combinations).

Additionally:

#UD	If VEX.W = 1.

VMAXPH—Return Maximum of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 5F /r VMAXPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Return the maximum packed FP16 values between xmm2 and xmm3/m128/m16bcst and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 5F /r VMAXPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Return the maximum packed FP16 values between ymm2 and ymm3/m256/m16bcst and store the result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 5F /r VMAXPH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {sae}	A	V/V	AVX512_FP16 OR AVX10.1	Return the maximum packed FP16 values between zmm2 and zmm3/m512/m16bcst and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a SIMD compare of the packed FP16 values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of VMAXPH can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcast from a 16-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Operation

```
def MAX(SRC1, SRC2):
    IF (SRC1 = 0.0) and (SRC2 = 0.0):
        DEST := SRC2
    ELSE IF (SRC1 = NaN):
        DEST := SRC2
    ELSE IF (SRC2 = NaN):
        DEST := SRC2
    ELSE IF (SRC1 > SRC2):
        DEST := SRC1
    ELSE:
        DEST := SRC2
```

VMAXPH dest, src1, src2

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            tsrc2 := SRC2.fp16[0]
        ELSE:
            tsrc2 := SRC2.fp16[j]
        DEST.fp16[j] := MAX(SRC1.fp16[j], tsrc2)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMAXPH __m128h _mm_mask_max_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMAXPH __m128h _mm_maskz_max_ph (__mmask8 k, __m128h a, __m128h b);
VMAXPH __m128h _mm_max_ph (__m128h a, __m128h b);
VMAXPH __m256h _mm256_mask_max_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VMAXPH __m256h _mm256_maskz_max_ph (__mmask16 k, __m256h a, __m256h b);
VMAXPH __m256h _mm256_max_ph (__m256h a, __m256h b);
VMAXPH __m512h _mm512_mask_max_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VMAXPH __m512h _mm512_maskz_max_ph (__mmask32 k, __m512h a, __m512h b);
VMAXPH __m512h _mm512_max_ph (__m512h a, __m512h b);
VMAXPH __m512h _mm512_mask_max_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int sae);
VMAXPH __m512h _mm512_maskz_max_round_ph (__mmask32 k, __m512h a, __m512h b, int sae);
VMAXPH __m512h _mm512_max_round_ph (__m512h a, __m512h b, int sae);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VMAXSH—Return Maximum of Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 5F /r VMAXSH xmm1{k1}{z}, xmm2, xmm3/m16 {sae}	A	V/V	AVX512_FP16 OR AVX10.1	Return the maximum low FP16 value between xmm3/m16 and xmm2 and store the result in xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a compare of the low packed FP16 values in the first source operand and the second source operand and returns the maximum value for the pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of VMAXSH can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

```
def MAX(SRC1, SRC2):
    IF (SRC1 = 0.0) and (SRC2 = 0.0):
        DEST := SRC2
    ELSE IF (SRC1 = NaN):
        DEST := SRC2
    ELSE IF (SRC2 = NaN):
        DEST := SRC2
    ELSE IF (SRC1 > SRC2):
        DEST := SRC1
    ELSE:
        DEST := SRC2
```

VMAXSH dest, src1, src2

```
IF k1[0] OR *no writemask*:
    DEST.fp16[0] := MAX(SRC1.fp16[0], SRC2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else dest.fp16[j] remains unchanged

DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VMAXSH __m128h _mm_mask_max_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int sae);
VMAXSH __m128h _mm_maskz_max_round_sh (__mmask8 k, __m128h a, __m128h b, int sae);
VMAXSH __m128h _mm_max_round_sh (__m128h a, __m128h b, int sae);
VMAXSH __m128h _mm_mask_max_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMAXSH __m128h _mm_maskz_max_sh (__mmask8 k, __m128h a, __m128h b);
VMAXSH __m128h _mm_max_sh (__m128h a, __m128h b);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VMINPH—Return Minimum of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 5D /r VMINPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Return the minimum packed FP16 values between xmm2 and xmm3/m128/m16bcst and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 5D /r VMINPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Return the minimum packed FP16 values between ymm2 and ymm3/m256/m16bcst and store the result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 5D /r VMINPH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {sae}	A	V/V	AVX512_FP16 OR AVX10.1	Return the minimum packed FP16 values between zmm2 and zmm3/m512/m16bcst and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a SIMD compare of the packed FP16 values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of VMINPH can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcast from a 16-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Operation

```
def MIN(SRC1, SRC2):
    IF (SRC1 = 0.0) and (SRC2 = 0.0):
        DEST := SRC2
    ELSE IF (SRC1 = NaN):
        DEST := SRC2
    ELSE IF (SRC2 = NaN):
        DEST := SRC2
    ELSE IF (SRC1 < SRC2):
        DEST := SRC1
    ELSE:
        DEST := SRC2
```

VMINPH dest, src1, src2

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            tsrc2 := SRC2.fp16[0]
        ELSE:
            tsrc2 := SRC2.fp16[j]
        DEST.fp16[j] := MIN(SRC1.fp16[j], tsrc2)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMINPH __m128h _mm_mask_min_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMINPH __m128h _mm_maskz_min_ph (__mmask8 k, __m128h a, __m128h b);
VMINPH __m128h _mm_min_ph (__m128h a, __m128h b);
VMINPH __m256h _mm256_mask_min_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VMINPH __m256h _mm256_maskz_min_ph (__mmask16 k, __m256h a, __m256h b);
VMINPH __m256h _mm256_min_ph (__m256h a, __m256h b);
VMINPH __m512h _mm512_mask_min_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VMINPH __m512h _mm512_maskz_min_ph (__mmask32 k, __m512h a, __m512h b);
VMINPH __m512h _mm512_min_ph (__m512h a, __m512h b);
VMINPH __m512h _mm512_mask_min_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int sae);
VMINPH __m512h _mm512_maskz_min_round_ph (__mmask32 k, __m512h a, __m512h b, int sae);
VMINPH __m512h _mm512_min_round_ph (__m512h a, __m512h b, int sae);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VMINSH—Return Minimum Scalar FP16 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 5D /r VMINSH xmm1{k1}{z}, xmm2, xmm3/m16 {sae}	A	V/V	AVX512_FP16 OR AVX10.1	Return the minimum low FP16 value between xmm3/m16 and xmm2. Stores the result in xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a compare of the low packed FP16 values in the first source operand and the second source operand and returns the minimum value for the pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of VMINSH can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcast from a 16-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

```
def MIN(SRC1, SRC2):
    IF (SRC1 = 0.0) and (SRC2 = 0.0):
        DEST := SRC2
    ELSE IF (SRC1 = NaN):
        DEST := SRC2
    ELSE IF (SRC2 = NaN):
        DEST := SRC2
    ELSE IF (SRC1 < SRC2):
        DEST := SRC1
    ELSE:
        DEST := SRC2
```

VMINSH dest, src1, src2

```
IF k1[0] OR *no writemask*:
    DEST.fp16[0] := MIN(SRC1.fp16[0], SRC2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else dest.fp16[j] remains unchanged

DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VMINSH __m128h _mm_mask_min_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int sae);
VMINSH __m128h _mm_maskz_min_round_sh (__mmask8 k, __m128h a, __m128h b, int sae);
VMINSH __m128h _mm_min_round_sh (__m128h a, __m128h b, int sae);
VMINSH __m128h _mm_mask_min_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMINSH __m128h _mm_maskz_min_sh (__mmask8 k, __m128h a, __m128h b);
VMINSH __m128h _mm_min_sh (__m128h a, __m128h b);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VMOVSH—Move Scalar FP16 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 10 /r VMOVSH xmm1{k1}{z}, m16	A	V/V	AVX512_FP16 OR AVX10.1	Move FP16 value from m16 to xmm1 subject to writemask k1.
EVEX.LLIG.F3.MAP5.W0 11 /r VMOVSH m16{k1}, xmm1	B	V/V	AVX512_FP16 OR AVX10.1	Move low FP16 value from xmm1 to m16 subject to writemask k1.
EVEX.LLIG.F3.MAP5.W0 10 /r VMOVSH xmm1{k1}{z}, xmm2, xmm3	C	V/V	AVX512_FP16 OR AVX10.1	Move low FP16 values from xmm3 to xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].
EVEX.LLIG.F3.MAP5.W0 11 /r VMOVSH xmm1{k1}{z}, xmm2, xmm3	D	V/V	AVX512_FP16 OR AVX10.1	Move low FP16 values from xmm3 to xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
C	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
D	N/A	ModRM:r/m (w)	VEX.vvvv (r)	ModRM:reg (r)	N/A

Description

This instruction moves a FP16 value to a register or memory location.

The two register-only forms are aliases and differ only in where their operands are encoded; this is a side effect of the encodings selected.

Operation

VMOVSH dest, src (two operand load)

```
IF k1[0] or no writemask:
    DEST.fp16[0] := SRC.fp16[0]
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// ELSE DEST.fp16[0] remains unchanged

DEST[MAXVL:16] := 0
```

VMOVSH dest, src (two operand store)

```
IF k1[0] or no writemask:
    DEST.fp16[0] := SRC.fp16[0]
// ELSE DEST.fp16[0] remains unchanged
```

VMOVSH dest, src1, src2 (three operand copy)

```
IF k1[0] or no writemask:
    DEST.fp16[0] := SRC2.fp16[0]
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// ELSE DEST.fp16[0] remains unchanged

DEST[127:16] := SRC1[127:16]
DEST[MAXVL:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVSH __m128h _mm_load_sh (void const* mem_addr);
VMOVSH __m128h _mm_mask_load_sh (__m128h src, __mmask8 k, void const* mem_addr);
VMOVSH __m128h _mm_maskz_load_sh (__mmask8 k, void const* mem_addr);
VMOVSH __m128h _mm_mask_move_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMOVSH __m128h _mm_maskz_move_sh (__mmask8 k, __m128h a, __m128h b);
VMOVSH __m128h _mm_move_sh (__m128h a, __m128h b);
VMOVSH void _mm_mask_store_sh (void * mem_addr, __mmask8 k, __m128h a);
VMOVSH void _mm_store_sh (void * mem_addr, __m128h a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Table 2-53, “Type E5 Class Exception Conditions.”

VMOVW—Move Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.WIG 6E /r VMOVW xmm1, reg/m16	A	V/V	AVX512_FP16 OR AVX10.1	Copy word from reg/m16 to xmm1.
EVEX.128.66.MAP5.WIG 7E /r VMOVW reg/m16, xmm1	B	V/V	AVX512_FP16 OR AVX10.1	Copy word from xmm1 to reg/m16.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

This instruction either (a) copies one word element from an XMM register to a general-purpose register or memory location or (b) copies one word element from a general-purpose register or memory location to an XMM register. When writing a general-purpose register, the lower 16-bits of the register will contain the word value. The upper bits of the general-purpose register are written with zeros.

Operation

VMOVW dest, src (two operand load)

```
DEST.word[0] := SRC.word[0]
DEST[MAXVL:16] := 0
```

VMOVW dest, src (two operand store)

```
DEST.word[0] := SRC.word[0]
// upper bits of GPR DEST are zeroed
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVW short _mm_cvtsi128_si16 (__m128i a);
VMOVW __m128i _mm_cvtsi16_si128 (short a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instructions, see Table 2-59, “Type E9NF Class Exception Conditions.”

VMULPH—Multiply Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 59 /r VMULPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm3/m128/m16bcst to xmm2 and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 59 /r VMULPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm3/m256/m16bcst to ymm2 and store the result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 59 /r VMULPH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply packed FP16 values in zmm3/m512/m16bcst with zmm2 and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction multiplies packed FP16 values from source operands and stores the packed FP16 result in the destination operand. The destination elements are updated according to the writemask.

Operation

VMULPH (EVEX encoded versions) when src2 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.fp16[j] := SRC1.fp16[j] * SRC2.fp16[j]
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VMULPH (EVEX encoded versions) when src2 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            DEST.fp16[j] := SRC1.fp16[j] * SRC2.fp16[0]
        ELSE:
            DEST.fp16[j] := SRC1.fp16[j] * SRC2.fp16[j]
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMULPH __m128h _mm_mask_mul_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMULPH __m128h _mm_maskz_mul_ph (__mmask8 k, __m128h a, __m128h b);
VMULPH __m128h _mm_mul_ph (__m128h a, __m128h b);
VMULPH __m256h _mm256_mask_mul_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VMULPH __m256h _mm256_maskz_mul_ph (__mmask16 k, __m256h a, __m256h b);
VMULPH __m256h _mm256_mul_ph (__m256h a, __m256h b);
VMULPH __m512h _mm512_mask_mul_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VMULPH __m512h _mm512_maskz_mul_ph (__mmask32 k, __m512h a, __m512h b);
VMULPH __m512h _mm512_mul_ph (__m512h a, __m512h b);
VMULPH __m512h _mm512_mask_mul_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int rounding);
VMULPH __m512h _mm512_maskz_mul_round_ph (__mmask32 k, __m512h a, __m512h b, int rounding);
VMULPH __m512h _mm512_mul_round_ph (__m512h a, __m512h b, int rounding);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VMULSH—Multiply Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 59 /r VMULSH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512_FP16 OR AVX10.1	Multiply the low FP16 value in xmm3/m16 by low FP16 value in xmm2, and store the result in xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction multiplies the low FP16 value from the source operands and stores the FP16 result in the destination operand. Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

VMULSH (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC2 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    DEST.fp16[0] := SRC1.fp16[0] * SRC2.fp16[0]
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else dest.fp16[0] remains unchanged

DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VMULSH __m128h _mm_mask_mul_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int rounding);
VMULSH __m128h _mm_maskz_mul_round_sh (__mmask8 k, __m128h a, __m128h b, int rounding);
VMULSH __m128h _mm_mul_round_sh (__m128h a, __m128h b, int rounding);
VMULSH __m128h _mm_mask_mul_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMULSH __m128h _mm_maskz_mul_sh (__mmask8 k, __m128h a, __m128h b);
VMULSH __m128h _mm_mul_sh (__m128h a, __m128h b);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VP2INTERSECTD/VP2INTERSECTQ—Compute Intersection Between DWORDS/QUADWORDS to a Pair of Mask Registers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.F2.0F38.W0 68 /r VP2INTERSECTD k1+1, xmm2, xmm3/m128/m32bcst	A	V/V	AVX512VL AVX512_VP2INTERSECT	Store, in an even/odd pair of mask registers, the indicators of the locations of value matches between dwords in xmm3/m128/m32bcst and xmm2.
EVEX.NDS.256.F2.0F38.W0 68 /r VP2INTERSECTD k1+1, ymm2, ymm3/m256/m32bcst	A	V/V	AVX512VL AVX512_VP2INTERSECT	Store, in an even/odd pair of mask registers, the indicators of the locations of value matches between dwords in ymm3/m256/m32bcst and ymm2.
EVEX.NDS.512.F2.0F38.W0 68 /r VP2INTERSECTD k1+1, zmm2, zmm3/m512/m32bcst	A	V/V	AVX512F AVX512_VP2INTERSECT	Store, in an even/odd pair of mask registers, the indicators of the locations of value matches between dwords in zmm3/m512/m32bcst and zmm2.
EVEX.NDS.128.F2.0F38.W1 68 /r VP2INTERSECTQ k1+1, xmm2, xmm3/m128/m64bcst	A	V/V	AVX512VL AVX512_VP2INTERSECT	Store, in an even/odd pair of mask registers, the indicators of the locations of value matches between quadwords in xmm3/m128/m64bcst and xmm2.
EVEX.NDS.256.F2.0F38.W1 68 /r VP2INTERSECTQ k1+1, ymm2, ymm3/m256/m64bcst	A	V/V	AVX512VL AVX512_VP2INTERSECT	Store, in an even/odd pair of mask registers, the indicators of the locations of value matches between quadwords in ymm3/m256/m64bcst and ymm2.
EVEX.NDS.512.F2.0F38.W1 68 /r VP2INTERSECTQ k1+1, zmm2, zmm3/m512/m64bcst	A	V/V	AVX512F AVX512_VP2INTERSECT	Store, in an even/odd pair of mask registers, the indicators of the locations of value matches between quadwords in zmm3/m512/m64bcst and zmm2.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction writes an even/odd pair of mask registers. The mask register destination indicated in the MODRM.REG field is used to form the basis of the register pair. The low bit of that field is masked off (set to zero) to create the first register of the pair.

EVEX.aaa and EVEX.z must be zero.

Operation

VP2INTERSECTD destmask, src1, src2
(KL, VL) = (4, 128), (8, 256), (16, 512)

```
// dest_mask_reg_id is the register id specified in the instruction for destmask
dest_base := dest_mask_reg_id & ~1

// maskregs[ ] is an array representing the mask registers
maskregs[dest_base+0][MAX_KL-1:0] := 0
maskregs[dest_base+1][MAX_KL-1:0] := 0

FOR i := 0 to KL-1:
    FOR j := 0 to KL-1:
        match := (src1.dword[i] == src2.dword[j])
        maskregs[dest_base+0].bit[i] |= match
        maskregs[dest_base+1].bit[j] |= match
```

VP2INTERSECTQ destmask, src1, src2
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
// dest_mask_reg_id is the register id specified in the instruction for destmask
dest_base := dest_mask_reg_id & ~1

// maskregs[ ] is an array representing the mask registers
maskregs[dest_base+0][MAX_KL-1:0] := 0
maskregs[dest_base+1][MAX_KL-1:0] := 0

FOR i = 0 to KL-1:
    FOR j = 0 to KL-1:
        match := (src1.qword[i] == src2.qword[j])
        maskregs[dest_base+0].bit[i] |=  match
        maskregs[dest_base+1].bit[j] |=  match
```

Intel C/C++ Compiler Intrinsic Equivalent

VP2INTERSECTD void _mm_2intersect_epi32(__m128i, __m128i, __mmask8 *, __mmask8 *);
VP2INTERSECTD void _mm256_2intersect_epi32(__m256i, __m256i, __mmask8 *, __mmask8 *);
VP2INTERSECTD void _mm512_2intersect_epi32(__m512i, __m512i, __mmask16 *, __mmask16 *);
VP2INTERSECTQ void _mm_2intersect_epi64(__m128i, __m128i, __mmask8 *, __mmask8 *);
VP2INTERSECTQ void _mm256_2intersect_epi64(__m256i, __m256i, __mmask8 *, __mmask8 *);
VP2INTERSECTQ void _mm512_2intersect_epi64(__m512i, __m512i, __mmask8 *, __mmask8 *);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-52, "Type E4NF Class Exception Conditions."

VPBLENDD—Blend Packed Dwords

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.128.66.0F3A.W0 02 /r ib VPBLENDD xmm1, xmm2, xmm3/m128, imm8	RVMI	V/V	AVX2	Select dwords from xmm2 and xmm3/m128 from mask specified in imm8 and store the values into xmm1.
VEX.256.66.0F3A.W0 02 /r ib VPBLENDD ymm1, ymm2, ymm3/m256, imm8	RVMI	V/V	AVX2	Select dwords from ymm2 and ymm3/m256 from mask specified in imm8 and store the values into ymm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Dword elements from the source operand (second operand) are conditionally written to the destination operand (first operand) depending on bits in the immediate operand (third operand). The immediate bits (bits 7:0) form a mask that determines whether the corresponding dword in the destination is copied from the source. If a bit in the mask, corresponding to a dword, is "1", then the dword is copied, else the dword 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 (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

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[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPBLENDD: __m128i _mm_blend_epi32 (__m128i v1, __m128i v2, const int mask)

VPBLENDD: __m256i _mm256_blend_epi32 (__m256i v1, __m256i v2, const int mask)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions."

Additionally:

#UD	If VEX.W = 1.

VPBLENDMB/VPBLENDMW—Blend Byte/Word Vectors Using an Opmask Control

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 66 /r VPBLENDMB xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Blend byte integer vector xmm2 and byte vector xmm3/m128 and store the result in xmm1, under control mask.
EVEX.256.66.0F38.W0 66 /r VPBLENDMB ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Blend byte integer vector ymm2 and byte vector ymm3/m256 and store the result in ymm1, under control mask.
EVEX.512.66.0F38.W0 66 /r VPBLENDMB zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512BW OR AVX10.1	Blend byte integer vector zmm2 and byte vector zmm3/m512 and store the result in zmm1, under control mask.
EVEX.128.66.0F38.W1 66 /r VPBLENDMW xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Blend word integer vector xmm2 and word vector xmm3/m128 and store the result in xmm1, under control mask.
EVEX.256.66.0F38.W1 66 /r VPBLENDMW ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Blend word integer vector ymm2 and word vector ymm3/m256 and store the result in ymm1, under control mask.
EVEX.512.66.0F38.W1 66 /r VPBLENDMW zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512BW OR AVX10.1	Blend word integer vector zmm2 and word vector zmm3/m512 and store the result in zmm1, under control mask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs an element-by-element blending of byte/word elements between the first source operand byte vector register and the second source operand byte vector from memory or register, using the instruction mask as selector. The result is written into the destination byte vector register.

The destination and first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit memory location.

The mask is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for first source, 1 for second source).

Operation

VPBLENDMB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)

```
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SRC2[i+7:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN DEST[i+7:i] := SRC1[i+7:i]
                ELSE                            ; zeroing-masking
                    DEST[i+7:i] := 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0;
```

VPBLENDMW (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)

```
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SRC2[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN DEST[i+15:i] := SRC1[i+15:i]
                ELSE                            ; zeroing-masking
                    DEST[i+15:i] := 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPBLENDMB __m512i _mm512_mask_blend_epi8(__mmask64 m, __m512i a, __m512i b);
VPBLENDMB __m256i _mm256_mask_blend_epi8(__mmask32 m, __m256i a, __m256i b);
VPBLENDMB __m128i _mm_mask_blend_epi8(__mmask16 m, __m128i a, __m128i b);
VPBLENDMW __m512i _mm512_mask_blend_epi16(__mmask32 m, __m512i a, __m512i b);
VPBLENDMW __m256i _mm256_mask_blend_epi16(__mmask16 m, __m256i a, __m256i b);
VPBLENDMW __m128i _mm_mask_blend_epi16(__mmask8 m, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-51, "Type E4 Class Exception Conditions."

VPBLENDMD/VPBLENDMQ—Blend Int32/Int64 Vectors Using an OpMask Control

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 64 /r VPBLENDMD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Blend doubleword integer vector xmm2 and doubleword vector xmm3/m128/m32bcst and store the result in xmm1, under control mask.
EVEX.256.66.0F38.W0 64 /r VPBLENDMD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Blend doubleword integer vector ymm2 and doubleword vector ymm3/m256/m32bcst and store the result in ymm1, under control mask.
EVEX.512.66.0F38.W0 64 /r VPBLENDMD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX512F OR AVX10.1	Blend doubleword integer vector zmm2 and doubleword vector zmm3/m512/m32bcst and store the result in zmm1, under control mask.
EVEX.128.66.0F38.W1 64 /r VPBLENDMQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Blend quadword integer vector xmm2 and quadword vector xmm3/m128/m64bcst and store the result in xmm1, under control mask.
EVEX.256.66.0F38.W1 64 /r VPBLENDMQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Blend quadword integer vector ymm2 and quadword vector ymm3/m256/m64bcst and store the result in ymm1, under control mask.
EVEX.512.66.0F38.W1 64 /r VPBLENDMQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	A	V/V	AVX512F OR AVX10.1	Blend quadword integer vector zmm2 and quadword vector zmm3/m512/m64bcst and store the result in zmm1, under control mask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs an element-by-element blending of dword/qword elements between the first source operand (the second operand) and the elements of the second source operand (the third operand) using an opmask register as select control. The blended result is written into the destination.

The destination and first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register; a 512/256/128-bit memory location; or a 512/256/128-bit vector with 16/8/4 identical values broadcast from a 32-bit memory location (VPBLENDMD) or 8/4/2 identical values broadcast from a 64-bit memory location (VPBLENDMQ).

The opmask register is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for the first source operand, 1 for the second source operand).

If EVEX.z is set, the elements with corresponding mask bit value of 0 in the destination operand are zeroed.

Operation

VPBLENDMD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no controlmask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] := SRC2[31:0]
                ELSE
                    DEST[i+31:i] := SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN DEST[i+31:i] := SRC1[i+31:i]
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0;
```

VPBLENDMQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no controlmask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63i] := SRC2[63:0]
                ELSE
                    DEST[i+63:i] := SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN DEST[i+63:i] := SRC1[i+63:i]
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPBLENDMD __m512i _mm512_mask_blend_epi32(__mmask16 k, __m512i a, __m512i b);
VPBLENDMD __m256i _mm256_mask_blend_epi32(__mmask8 m, __m256i a, __m256i b);
VPBLENDMD __m128i _mm_mask_blend_epi32(__mmask8 m, __m128i a, __m128i b);
VPBLENDMQ __m512i _mm512_mask_blend_epi64(__mmask8 k, __m512i a, __m512i b);
VPBLENDMQ __m256i _mm256_mask_blend_epi64(__mmask8 m, __m256i a, __m256i b);
VPBLENDMQ __m128i _mm_mask_blend_epi64(__mmask8 m, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-51, "Type E4 Class Exception Conditions."

VPBROADCAST—Load Integer and Broadcast

Opcode/ Instruction	Op / En	64/32 bit Mode Support	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.
EVEX.128.66.0F38.W0 78 /r VPBROADCASTB xmm1{k1}{z}, xmm2/m8	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Broadcast a byte integer in the source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 78 /r VPBROADCASTB ymm1{k1}{z}, xmm2/m8	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Broadcast a byte integer in the source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 78 /r VPBROADCASTB zmm1{k1}{z}, xmm2/m8	B	V/V	AVX512BW OR AVX10.1	Broadcast a byte integer in the source operand to 64 locations in zmm1 subject to writemask k1.
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.
EVEX.128.66.0F38.W0 79 /r VPBROADCASTW xmm1{k1}{z}, xmm2/m16	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Broadcast a word integer in the source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 79 /r VPBROADCASTW ymm1{k1}{z}, xmm2/m16	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Broadcast a word integer in the source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 79 /r VPBROADCASTW zmm1{k1}{z}, xmm2/m16	B	V/V	AVX512BW OR AVX10.1	Broadcast a word integer in the source operand to 32 locations in zmm1 subject to writemask k1.
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.
EVEX.128.66.0F38.W0 58 /r VPBROADCASTD xmm1 {k1}{z}, xmm2/m32	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Broadcast a dword integer in the source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 58 /r VPBROADCASTD ymm1 {k1}{z}, xmm2/m32	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Broadcast a dword integer in the source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 58 /r VPBROADCASTD zmm1 {k1}{z}, xmm2/m32	B	V/V	AVX512F OR AVX10.1	Broadcast a dword integer in the source operand to locations in zmm1 subject to writemask k1.
VEX.128.66.0F38.W0 59 /r VPBROADCASTQ xmm1, xmm2/m64	A	V/V	AVX2	Broadcast a qword element in source operand 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 source operand to four locations in ymm1.
EVEX.128.66.0F38.W1 59 /r VPBROADCASTQ xmm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Broadcast a qword element in source operand to locations in xmm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F38.W1 59 /r VPBROADCASTQ ymm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Broadcast a qword element in source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W1 59 /r VPBROADCASTQ zmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512F OR AVX10.1	Broadcast a qword element in source operand to locations in zmm1 subject to writemask k1.
EVEX.128.66.0F38.W0 59 /r VBROADCASTI32x2 xmm1 {k1}{z}, xmm2/m64	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Broadcast two dword elements in source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 59 /r VBROADCASTI32x2 ymm1 {k1}{z}, xmm2/m64	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Broadcast two dword elements in source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 59 /r VBROADCASTI32x2 zmm1 {k1}{z}, xmm2/m64	C	V/V	AVX512DQ OR AVX10.1	Broadcast two dword elements in source operand to locations in zmm1 subject to writemask k1.
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.
EVEX.256.66.0F38.W0 5A /r VBROADCASTI32X4 ymm1 {k1}{z}, m128	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Broadcast 128 bits of 4 doubleword integer data in mem to locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 5A /r VBROADCASTI32X4 zmm1 {k1}{z}, m128	D	V/V	AVX512F OR AVX10.1	Broadcast 128 bits of 4 doubleword integer data in mem to locations in zmm1 using writemask k1.
EVEX.256.66.0F38.W1 5A /r VBROADCASTI64X2 ymm1 {k1}{z}, m128	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Broadcast 128 bits of 2 quadword integer data in mem to locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 5A /r VBROADCASTI64X2 zmm1 {k1}{z}, m128	C	V/V	AVX512DQ OR AVX10.1	Broadcast 128 bits of 2 quadword integer data in mem to locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W0 5B /r VBROADCASTI32X8 zmm1 {k1}{z}, m256	E	V/V	AVX512DQ OR AVX10.1	Broadcast 256 bits of 8 doubleword integer data in mem to locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W1 5B /r VBROADCASTI64X4 zmm1 {k1}{z}, m256	D	V/V	AVX512F OR AVX10.1	Broadcast 256 bits of 4 quadword integer data in mem to locations in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
C	Tuple2	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
D	Tuple4	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
E	Tuple8	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Load integer data from the source operand (the second operand) and broadcast to all elements of the destination operand (the first operand).

VEX256-encoded VPBROADCASTB/W/D/Q: The source operand is 8-bit, 16-bit, 32-bit, 64-bit memory location or the low 8-bit, 16-bit 32-bit, 64-bit data in an XMM register. The destination operand is a YMM register. VPBROADCASTI128 support the source operand of 128-bit memory location. Register source encodings for VPBROADCASTI128 is reserved and will #UD. Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX-encoded VPBROADCASTD/Q: The source operand is a 32-bit, 64-bit memory location or the low 32-bit, 64-bit data in an XMM register. The destination operand is a ZMM/YMM/XMM register and updated according to the writemask k1.

VPBROADCASTI32X4 and VPBROADCASTI64X4: The destination operand is a ZMM register and updated according to the writemask k1. The source operand is 128-bit or 256-bit memory location. Register source encodings for VBROADCASTI32X4 and VBROADCASTI64X4 are reserved and will #UD.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

If VPBROADCASTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

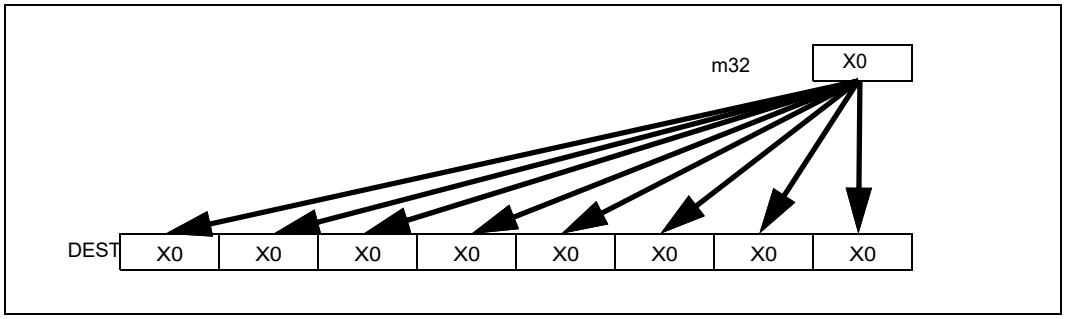


Figure 5-16. VPBROADCASTD Operation (VEX.256 encoded version)

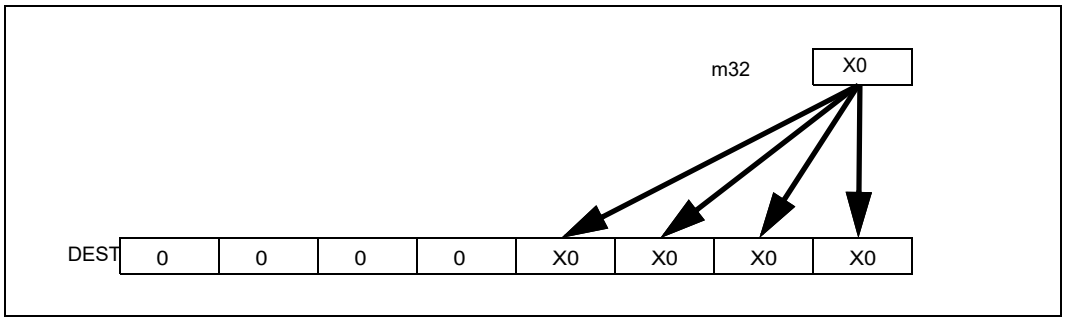


Figure 5-17. VPBROADCASTD Operation (128-bit version)

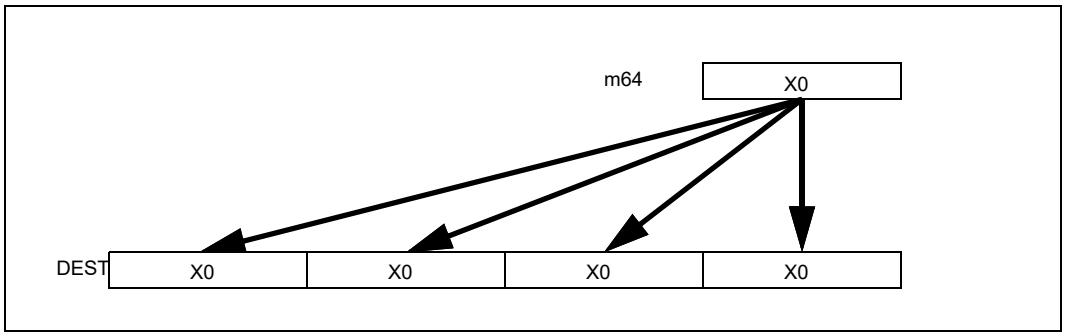


Figure 5-18. VPBROADCASTQ Operation (256-bit version)

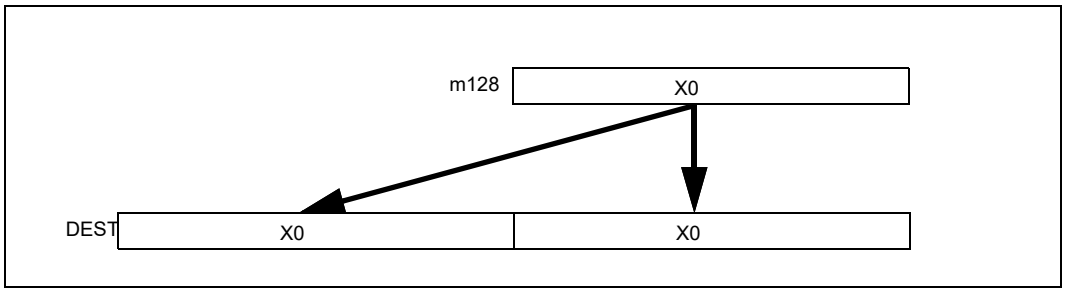


Figure 5-19. VBROADCASTI128 Operation (256-bit version)

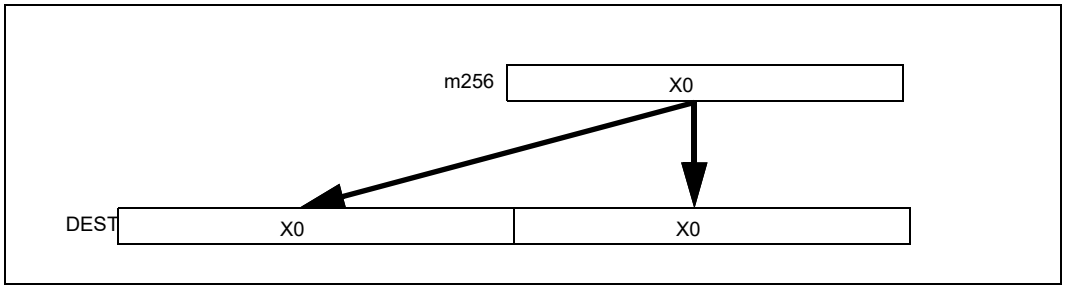


Figure 5-20. VBROADCASTI256 Operation (512-bit version)

Operation

VPBROADCASTB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SRC[7:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPBROADCASTW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SRC[15:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPBROADCASTD (128 bit version)

```
temp := SRC[31:0]
DEST[31:0] := temp
DEST[63:32] := temp
DEST[95:64] := temp
DEST[127:96] := temp
DEST[MAXVL-1:128] := 0
```

VPBROADCASTD (VEX.256 encoded version)

```
temp := SRC[31:0]
DEST[31:0] := temp
DEST[63:32] := temp
DEST[95:64] := temp
DEST[127:96] := temp
DEST[159:128] := temp
DEST[191:160] := temp
DEST[223:192] := temp
DEST[255:224] := temp
DEST[MAXVL-1:256] := 0
```

VPBROADCASTD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[31:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPBROADCASTQ (VEX.256 encoded version)

```
temp := SRC[63:0]
DEST[63:0] := temp
DEST[127:64] := temp
DEST[191:128] := temp
DEST[255:192] := temp
DEST[MAXVL-1:256] := 0
```

VPBROADCASTQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[63:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VBROADCASTI32x2 (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    n := (j mod 2) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[n+31:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
```

```
DEST[MAXVL-1:VL] := 0
```

VBROADCASTI128 (VEX.256 encoded version)

```
temp := SRC[127:0]
DEST[127:0] := temp
DEST[255:128] := temp
DEST[MAXVL-1:256] := 0
```

VBROADCASTI32X4 (EVEX encoded versions)

```
(KL, VL) = (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j* 32
    n := (j modulo 4) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[n+31:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VBROADCASTI64X2 (EVEX encoded versions)

```
(KL, VL) = (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 64
    n := (j modulo 2) * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[n+63:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] = 0
            FI
    FI;
ENDFOR;
```

VBROADCASTI32X8 (EVEX.U1.512 encoded version)

```
FOR j := 0 TO 15
    i := j * 32
    n := (j modulo 8) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[n+31:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
```

```
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VBROADCASTI64X4 (EVEX.512 encoded version)

```
FOR j := 0 TO 7
    i := j * 64
    n := (j modulo 4) * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[n+63:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPBROADCASTB __m512i _mm512_broadcastb_epi8(__m128i a);
VPBROADCASTB __m512i _mm512_mask_broadcastb_epi8(__m512i s, __mmask64 k, __m128i a);
VPBROADCASTB __m512i _mm512_maskz_broadcastb_epi8(__mmask64 k, __m128i a);
VPBROADCASTB __m256i _mm256_broadcastb_epi8(__m128i a);
VPBROADCASTB __m256i _mm256_mask_broadcastb_epi8(__m256i s, __mmask32 k, __m128i a);
VPBROADCASTB __m256i _mm256_maskz_broadcastb_epi8(__mmask32 k, __m128i a);
VPBROADCASTB __m128i _mm_mask_broadcastb_epi8(__m128i s, __mmask16 k, __m128i a);
VPBROADCASTB __m128i _mm_maskz_broadcastb_epi8(__mmask16 k, __m128i a);
VPBROADCASTB __m128i _mm_broadcastb_epi8(__m128i a);
VPBROADCASTD __m512i _mm512_broadcastd_epi32(__m128i a);
VPBROADCASTD __m512i _mm512_mask_broadcastd_epi32(__m512i s, __mmask16 k, __m128i a);
VPBROADCASTD __m512i _mm512_maskz_broadcastd_epi32(__mmask16 k, __m128i a);
VPBROADCASTD __m256i _mm256_broadcastd_epi32(__m128i a);
VPBROADCASTD __m256i _mm256_mask_broadcastd_epi32(__m256i s, __mmask8 k, __m128i a);
VPBROADCASTD __m256i _mm256_maskz_broadcastd_epi32(__mmask8 k, __m128i a);
VPBROADCASTD __m128i _mm_broadcastd_epi32(__m128i a);
VPBROADCASTD __m128i _mm_mask_broadcastd_epi32(__m128i s, __mmask8 k, __m128i a);
VPBROADCASTD __m128i _mm_maskz_broadcastd_epi32(__mmask8 k, __m128i a);
VPBROADCASTQ __m512i _mm512_broadcastq_epi64(__m128i a);
VPBROADCASTQ __m512i _mm512_mask_broadcastq_epi64(__m512i s, __mmask8 k, __m128i a);
VPBROADCASTQ __m512i _mm512_maskz_broadcastq_epi64(__mmask8 k, __m128i a);
VPBROADCASTQ __m256i _mm256_broadcastq_epi64(__m128i a);
VPBROADCASTQ __m256i _mm256_mask_broadcastq_epi64(__m256i s, __mmask8 k, __m128i a);
VPBROADCASTQ __m256i _mm256_maskz_broadcastq_epi64(__mmask8 k, __m128i a);
VPBROADCASTQ __m128i _mm_broadcastq_epi64(__m128i a);
VPBROADCASTQ __m128i _mm_mask_broadcastq_epi64(__m128i s, __mmask8 k, __m128i a);
VPBROADCASTQ __m128i _mm_maskz_broadcastq_epi64(__mmask8 k, __m128i a);
VPBROADCASTW __m512i _mm512_broadcastw_epi16(__m128i a);
VPBROADCASTW __m512i _mm512_mask_broadcastw_epi16(__m512i s, __mmask32 k, __m128i a);
VPBROADCASTW __m512i _mm512_maskz_broadcastw_epi16(__mmask32 k, __m128i a);
VPBROADCASTW __m256i _mm256_broadcastw_epi16(__m128i a);
VPBROADCASTW __m256i _mm256_mask_broadcastw_epi16(__m256i s, __mmask16 k, __m128i a);
VPBROADCASTW __m256i _mm256_maskz_broadcastw_epi16(__mmask16 k, __m128i a);
VPBROADCASTW __m128i _mm_broadcastw_epi16(__m128i a);

VPBROADCASTW __m128i _mm_mask_broadcastw_epi16(__m128i s, __mmask8 k, __m128i a);
VPBROADCASTW __m128i _mm_maskz_broadcastw_epi16(__mmask8 k, __m128i a);
VBROADCASTI32x2 __m512i _mm512_broadcast_i32x2(__m128i a);
VBROADCASTI32x2 __m512i _mm512_mask_broadcast_i32x2(__m512i s, __mmask16 k, __m128i a);
VBROADCASTI32x2 __m512i _mm512_maskz_broadcast_i32x2(__mmask16 k, __m128i a);
VBROADCASTI32x2 __m256i _mm256_broadcast_i32x2(__m128i a);
VBROADCASTI32x2 __m256i _mm256_mask_broadcast_i32x2(__m256i s, __mmask8 k, __m128i a);
VBROADCASTI32x2 __m256i _mm256_maskz_broadcast_i32x2(__mmask8 k, __m128i a);
VBROADCASTI32x2 __m128i _mm_broadcast_i32x2(__m128i a);
VBROADCASTI32x2 __m128i _mm_mask_broadcast_i32x2(__m128i s, __mmask8 k, __m128i a);
VBROADCASTI32x2 __m128i _mm_maskz_broadcast_i32x2(__mmask8 k, __m128i a);
VBROADCASTI32x4 __m512i _mm512_broadcast_i32x4(__m128i a);
VBROADCASTI32x4 __m512i _mm512_mask_broadcast_i32x4(__m512i s, __mmask16 k, __m128i a);
VBROADCASTI32x4 __m512i _mm512_maskz_broadcast_i32x4(__mmask16 k, __m128i a);
VBROADCASTI32x4 __m256i _mm256_broadcast_i32x4(__m128i a);
VBROADCASTI32x4 __m256i _mm256_mask_broadcast_i32x4(__m256i s, __mmask8 k, __m128i a);
VBROADCASTI32x4 __m256i _mm256_maskz_broadcast_i32x4(__mmask8 k, __m128i a);
VBROADCASTI32x8 __m512i _mm512_broadcast_i32x8(__m256i a);
VBROADCASTI32x8 __m512i _mm512_mask_broadcast_i32x8(__m512i s, __mmask16 k, __m256i a);
VBROADCASTI32x8 __m512i _mm512_maskz_broadcast_i32x8(__mmask16 k, __m256i a);
VBROADCASTI64x2 __m512i _mm512_broadcast_i64x2(__m128i a);
VBROADCASTI64x2 __m512i _mm512_mask_broadcast_i64x2(__m512i s, __mmask8 k, __m128i a);
VBROADCASTI64x2 __m512i _mm512_maskz_broadcast_i64x2(__mmask8 k, __m128i a);
VBROADCASTI64x2 __m256i _mm256_broadcast_i64x2(__m128i a);
VBROADCASTI64x2 __m256i _mm256_mask_broadcast_i64x2(__m256i s, __mmask8 k, __m128i a);
VBROADCASTI64x2 __m256i _mm256_maskz_broadcast_i64x2(__mmask8 k, __m128i a);
VBROADCASTI64x4 __m512i _mm512_broadcast_i64x4(__m256i a);
VBROADCASTI64x4 __m512i _mm512_mask_broadcast_i64x4(__m512i s, __mmask8 k, __m256i a);
VBROADCASTI64x4 __m512i _mm512_maskz_broadcast_i64x4(__mmask8 k, __m256i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instructions, see Table 2-23, "Type 6 Class Exception Conditions."
EVEX-encoded instructions, syntax with reg/mem operand, see Table 2-55, "Type E6 Class Exception Conditions."
Additionally:

#UD	If VEX.L = 0 for VPBROADCASTQ, VPBROADCASTI128.
	If EVEX.L'L = 0 for VBROADCASTI32X4/VBROADCASTI64X2.
	If EVEX.L'L < 10b for VBROADCASTI32X8/VBROADCASTI64X4.

VPBROADCASTB/W/D/Q—Load With Broadcast Integer Data From General Purpose Register

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 7A /r VPBROADCASTB xmm1 {k1}{z}, reg	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Broadcast an 8-bit value from a GPR to all bytes in the 128-bit destination subject to writemask k1.
EVEX.256.66.0F38.W0 7A /r VPBROADCASTB ymm1 {k1}{z}, reg	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Broadcast an 8-bit value from a GPR to all bytes in the 256-bit destination subject to writemask k1.
EVEX.512.66.0F38.W0 7A /r VPBROADCASTB zmm1 {k1}{z}, reg	A	V/V	AVX512BW OR AVX10.1	Broadcast an 8-bit value from a GPR to all bytes in the 512-bit destination subject to writemask k1.
EVEX.128.66.0F38.W0 7B /r VPBROADCASTW xmm1 {k1}{z}, reg	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Broadcast a 16-bit value from a GPR to all words in the 128-bit destination subject to writemask k1.
EVEX.256.66.0F38.W0 7B /r VPBROADCASTW ymm1 {k1}{z}, reg	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Broadcast a 16-bit value from a GPR to all words in the 256-bit destination subject to writemask k1.
EVEX.512.66.0F38.W0 7B /r VPBROADCASTW zmm1 {k1}{z}, reg	A	V/V	AVX512BW OR AVX10.1	Broadcast a 16-bit value from a GPR to all words in the 512-bit destination subject to writemask k1.
EVEX.128.66.0F38.W0 7C /r VPBROADCASTD xmm1 {k1}{z}, r32	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Broadcast a 32-bit value from a GPR to all doublewords in the 128-bit destination subject to writemask k1.
EVEX.256.66.0F38.W0 7C /r VPBROADCASTD ymm1 {k1}{z}, r32	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Broadcast a 32-bit value from a GPR to all doublewords in the 256-bit destination subject to writemask k1.
EVEX.512.66.0F38.W0 7C /r VPBROADCASTD zmm1 {k1}{z}, r32	A	V/V	AVX512F OR AVX10.1	Broadcast a 32-bit value from a GPR to all doublewords in the 512-bit destination subject to writemask k1.
EVEX.128.66.0F38.W1 7C /r VPBROADCASTQ xmm1 {k1}{z}, r64	A	V/N.E.[1]	(AVX512VL AND AVX512F) OR AVX10.1	Broadcast a 64-bit value from a GPR to all quadwords in the 128-bit destination subject to writemask k1.
EVEX.256.66.0F38.W1 7C /r VPBROADCASTQ ymm1 {k1}{z}, r64	A	V/N.E.[1]	(AVX512VL AND AVX512F) OR AVX10.1	Broadcast a 64-bit value from a GPR to all quadwords in the 256-bit destination subject to writemask k1.
EVEX.512.66.0F38.W1 7C /r VPBROADCASTQ zmm1 {k1}{z}, r64	A	V/N.E.[1]	AVX512F OR AVX10.1	Broadcast a 64-bit value from a GPR to all quadwords in the 512-bit destination subject to writemask k1.

NOTES:

1. EVEX.W in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Broadcasts a 8-bit, 16-bit, 32-bit or 64-bit value from a general-purpose register (the second operand) to all the locations in the destination vector register (the first operand) using the writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPBROADCASTB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SRC[7:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPBROADCASTW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SRC[15:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPBROADCASTD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[31:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPBROADCASTQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[63:0]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPBROADCASTB __m512i _mm512_mask_set1_epi8(__m512i s, __mmask64 k, int a);
VPBROADCASTB __m512i _mm512_maskz_set1_epi8( __mmask64 k, int a);
VPBROADCASTB __m256i _mm256_mask_set1_epi8(__m256i s, __mmask32 k, int a);
VPBROADCASTB __m256i _mm256_maskz_set1_epi8( __mmask32 k, int a);
VPBROADCASTB __m128i _mm_mask_set1_epi8(__m128i s, __mmask16 k, int a);
VPBROADCASTB __m128i _mm_maskz_set1_epi8( __mmask16 k, int a);
VPBROADCASTD __m512i _mm512_mask_set1_epi32(__m512i s, __mmask16 k, int a);
VPBROADCASTD __m512i _mm512_maskz_set1_epi32( __mmask16 k, int a);
VPBROADCASTD __m256i _mm256_mask_set1_epi32(__m256i s, __mmask8 k, int a);
VPBROADCASTD __m256i _mm256_maskz_set1_epi32( __mmask8 k, int a);
VPBROADCASTD __m128i _mm_mask_set1_epi32(__m128i s, __mmask8 k, int a);
VPBROADCASTD __m128i _mm_maskz_set1_epi32( __mmask8 k, int a);
VPBROADCASTQ __m512i _mm512_mask_set1_epi64(__m512i s, __mmask8 k, __int64 a);
VPBROADCASTQ __m512i _mm512_maskz_set1_epi64( __mmask8 k, __int64 a);
VPBROADCASTQ __m256i _mm256_mask_set1_epi64(__m256i s, __mmask8 k, __int64 a);
VPBROADCASTQ __m256i _mm256_maskz_set1_epi64( __mmask8 k, __int64 a);
VPBROADCASTQ __m128i _mm_mask_set1_epi64(__m128i s, __mmask8 k, __int64 a);
VPBROADCASTQ __m128i _mm_maskz_set1_epi64( __mmask8 k, __int64 a);
VPBROADCASTW __m512i _mm512_mask_set1_epi16(__m512i s, __mmask32 k, int a);
VPBROADCASTW __m512i _mm512_maskz_set1_epi16( __mmask32 k, int a);
VPBROADCASTW __m256i _mm256_mask_set1_epi16(__m256i s, __mmask16 k, int a);
VPBROADCASTW __m256i _mm256_maskz_set1_epi16( __mmask16 k, int a);
VPBROADCASTW __m128i _mm_mask_set1_epi16(__m128i s, __mmask8 k, int a);
VPBROADCASTW __m128i _mm_maskz_set1_epi16( __mmask8 k, int a);
```

Exceptions

EVEX-encoded instructions, see Table 2-57, “Type E7NM Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VPBROADCASTM—Broadcast Mask to Vector Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W1 2A /r VPBROADCASTMB2Q xmm1, k1	RM	V/V	(AVX512VL AND AVX512CD) OR AVX10.1	Broadcast low byte value in k1 to two locations in xmm1.
EVEX.256.F3.0F38.W1 2A /r VPBROADCASTMB2Q ymm1, k1	RM	V/V	(AVX512VL AND AVX512CD) OR AVX10.1	Broadcast low byte value in k1 to four locations in ymm1.
EVEX.512.F3.0F38.W1 2A /r VPBROADCASTMB2Q zmm1, k1	RM	V/V	AVX512CD OR AVX10.1	Broadcast low byte value in k1 to eight locations in zmm1.
EVEX.128.F3.0F38.W0 3A /r VPBROADCASTMW2D xmm1, k1	RM	V/V	(AVX512VL AND AVX512CD) OR AVX10.1	Broadcast low word value in k1 to four locations in xmm1.
EVEX.256.F3.0F38.W0 3A /r VPBROADCASTMW2D ymm1, k1	RM	V/V	(AVX512VL AND AVX512CD) OR AVX10.1	Broadcast low word value in k1 to eight locations in ymm1.
EVEX.512.F3.0F38.W0 3A /r VPBROADCASTMW2D zmm1, k1	RM	V/V	AVX512CD OR AVX10.1	Broadcast low word value in k1 to sixteen locations in zmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Broadcasts the zero-extended 64/32 bit value of the low byte/word of the source operand (the second operand) to each 64/32 bit element of the destination operand (the first operand). The source operand is an opmask register. The destination operand is a ZMM register (EVEX.512), YMM register (EVEX.256), or XMM register (EVEX.128).

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPBROADCASTMB2Q

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j*64
    DEST[i+63:i] := ZeroExtend(SRC[7:0])
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPBROADCASTMW2D

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j*32
    DEST[i+31:i] := ZeroExtend(SRC[15:0])
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPBROADCASTMB2Q __m512i _mm512_broadcastmb_epi64(__mmask8);
VPBROADCASTMW2D __m512i _mm512_broadcastmw_epi32(__mmask16);
VPBROADCASTMB2Q __m256i _mm256_broadcastmb_epi64(__mmask8);
VPBROADCASTMW2D __m256i _mm256_broadcastmw_epi32(__mmask8);
VPBROADCASTMB2Q __m128i _mm_broadcastmb_epi64(__mmask8);
VPBROADCASTMW2D __m128i _mm_broadcastmw_epi32(__mmask8);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Table 2-56, “Type E6NF Class Exception Conditions.”

VPCMPB/VPCMPUB—Compare Packed Byte Values Into Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 3F /r ib VPCMPB k1 {k2}, xmm2, xmm3/m128, imm8	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed byte values in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.256.66.0F3A.W0 3F /r ib VPCMPB k1 {k2}, ymm2, ymm3/m256, imm8	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed byte values in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.512.66.0F3A.W0 3F /r ib VPCMPB k1 {k2}, zmm2, zmm3/m512, imm8	A	V/V	AVX512BW OR AVX10.1	Compare packed signed byte values in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.128.66.0F3A.W0 3E /r ib VPCMPUB k1 {k2}, xmm2, xmm3/m128, imm8	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed unsigned byte values in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.256.66.0F3A.W0 3E /r ib VPCMPUB k1 {k2}, ymm2, ymm3/m256, imm8	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed unsigned byte values in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.512.66.0F3A.W0 3E /r ib VPCMPUB k1 {k2}, zmm2, zmm3/m512, imm8	A	V/V	AVX512BW OR AVX10.1	Compare packed unsigned byte values in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed byte values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPB performs a comparison between pairs of signed byte values.

VPCMPUB performs a comparison between pairs of unsigned byte values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand (first operand) is a mask register k1. Up to 64/32/16 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-19.

Table 5-19. Pseudo-Op and VPCMP* Implementation

Pseudo-Op	PCMPM Implementation
VPCMPEQ* *reg1, reg2, reg3*	VPCMP* *reg1, reg2, reg3, 0*
VPCMPLT* *reg1, reg2, reg3*	VPCMP**reg1, reg2, reg3, 1*
VPCMPLE* *reg1, reg2, reg3*	VPCMP* *reg1, reg2, reg3, 2*
VPCMPNEQ* *reg1, reg2, reg3*	VPCMP* *reg1, reg2, reg3, 4*
VPPCMPNLT* *reg1, reg2, reg3*	VPCMP* *reg1, reg2, reg3, 5*
VPCMPNLE* *reg1, reg2, reg3*	VPCMP* *reg1, reg2, reg3, 6*

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP := EQ;
    1: OP := LT;
    2: OP := LE;
    3: OP := FALSE;
    4: OP := NEQ;
    5: OP := NLT;
    6: OP := NLE;
    7: OP := TRUE;
ESAC;
```

VPCMPB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k2[j] OR *no writemask*
        THEN
            CMP := SRC1[i+7:i] OP SRC2[i+7:i];
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] = 0                    ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPCMPUB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k2[j] OR *no writemask*
        THEN
            CMP := SRC1[i+7:i] OP SRC2[i+7:i];
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] = 0                     ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCMPB __mmask64 _mm512_cmp_epi8_mask(__m512i a, __m512i b, int cmp);
VPCMPB __mmask64 _mm512_mask_cmp_epi8_mask(__mmask64 m, __m512i a, __m512i b, int cmp);
VPCMPB __mmask32 _mm256_cmp_epi8_mask(__m256i a, __m256i b, int cmp);
VPCMPB __mmask32 _mm256_mask_cmp_epi8_mask(__mmask32 m, __m256i a, __m256i b, int cmp);
VPCMPB __mmask16 _mm_cmp_epi8_mask(__m128i a, __m128i b, int cmp);
VPCMPB __mmask16 _mm_mask_cmp_epi8_mask(__mmask16 m, __m128i a, __m128i b, int cmp);
VPCMPB __mmask64 _mm512_cmp[eq|ge|gt|le|lt|neq]_epi8_mask(__m512i a, __m512i b);
VPCMPB __mmask64 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi8_mask(__mmask64 m, __m512i a, __m512i b);
VPCMPB __mmask32 _mm256_cmp[eq|ge|gt|le|lt|neq]_epi8_mask(__m256i a, __m256i b);
VPCMPB __mmask32 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi8_mask(__mmask32 m, __m256i a, __m256i b);
VPCMPB __mmask16 _mm_cmp[eq|ge|gt|le|lt|neq]_epi8_mask(__m128i a, __m128i b);
VPCMPB __mmask16 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi8_mask(__mmask16 m, __m128i a, __m128i b);
VPCMPUB __mmask64 _mm512_cmp_epu8_mask(__m512i a, __m512i b, int cmp);
VPCMPUB __mmask64 _mm512_mask_cmp_epu8_mask(__mmask64 m, __m512i a, __m512i b, int cmp);
VPCMPUB __mmask32 _mm256_cmp_epu8_mask(__m256i a, __m256i b, int cmp);
VPCMPUB __mmask32 _mm256_mask_cmp_epu8_mask(__mmask32 m, __m256i a, __m256i b, int cmp);
VPCMPUB __mmask16 _mm_cmp_epu8_mask(__m128i a, __m128i b, int cmp);
VPCMPUB __mmask16 _mm_mask_cmp_epu8_mask(__mmask16 m, __m128i a, __m128i b, int cmp);
VPCMPUB __mmask64 _mm512_cmp[eq|ge|gt|le|lt|neq]_epu8_mask(__m512i a, __m512i b, int cmp);
VPCMPUB __mmask64 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu8_mask(__mmask64 m, __m512i a, __m512i b, int cmp);
VPCMPUB __mmask32 _mm256_cmp[eq|ge|gt|le|lt|neq]_epu8_mask(__m256i a, __m256i b, int cmp);
VPCMPUB __mmask32 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu8_mask(__mmask32 m, __m256i a, __m256i b, int cmp);
VPCMPUB __mmask16 _mm_cmp[eq|ge|gt|le|lt|neq]_epu8_mask(__m128i a, __m128i b, int cmp);
VPCMPUB __mmask16 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu8_mask(__mmask16 m, __m128i a, __m128i b, int cmp);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

VPCMPD/VPCMPUD—Compare Packed Integer Values Into Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 1F /r ib VPCMPD k1 {k2}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed signed doubleword integer values in xmm3/m128/m32bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.256.66.0F3A.W0 1F /r ib VPCMPD k1 {k2}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed signed doubleword integer values in ymm3/m256/m32bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.512.66.0F3A.W0 1F /r ib VPCMPD k1 {k2}, zmm2, zmm3/m512/m32bcst, imm8	A	V/V	AVX512F OR AVX10.1	Compare packed signed doubleword integer values in zmm2 and zmm3/m512/m32bcst using bits 2:0 of imm8 as a comparison predicate. The comparison results are written to the destination k1 under writemask k2.
EVEX.128.66.0F3A.W0 1E /r ib VPCMPUD k1 {k2}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed unsigned doubleword integer values in xmm3/m128/m32bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.256.66.0F3A.W0 1E /r ib VPCMPUD k1 {k2}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed unsigned doubleword integer values in ymm3/m256/m32bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.512.66.0F3A.W0 1E /r ib VPCMPUD k1 {k2}, zmm2, zmm3/m512/m32bcst, imm8	A	V/V	AVX512F OR AVX10.1	Compare packed unsigned doubleword integer values in zmm2 and zmm3/m512/m32bcst using bits 2:0 of imm8 as a comparison predicate. The comparison results are written to the destination k1 under writemask k2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Performs a SIMD compare of the packed integer values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPD/VPCMPUD performs a comparison between pairs of signed/unsigned doubleword integer values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is a mask register k1. Up to 16/8/4 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-19.

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP := EQ;
```

```
    1: OP := LT;
    2: OP := LE;
    3: OP := FALSE;
    4: OP := NEQ;
    5: OP := NLT;
    6: OP := NLE;
    7: OP := TRUE;
ESAC;
```

VPCMPD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+31:i] OP SRC2[31:0];
                ELSE CMP := SRC1[i+31:i] OP SRC2[i+31:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0                    ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPCMPUD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+31:i] OP SRC2[31:0];
                ELSE CMP := SRC1[i+31:i] OP SRC2[i+31:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0                    ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCMPD __mmask16 _mm512_cmp_epi32_mask(__m512i a, __m512i b, int imm);
VPCMPD __mmask16 _mm512_mask_cmp_epi32_mask(__mmask16 k, __m512i a, __m512i b, int imm);
VPCMPD __mmask16 _mm512_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__m512i a, __m512i b);
VPCMPD __mmask16 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPUD __mmask16 _mm512_cmp_epu32_mask(__m512i a, __m512i b, int imm);
VPCMPUD __mmask16 _mm512_mask_cmp_epu32_mask(__mmask16 k, __m512i a, __m512i b, int imm);
VPCMPUD __mmask16 _mm512_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__m512i a, __m512i b);
VPCMPUD __mmask16 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPD __mmask8 _mm256_cmp_epi32_mask(__m256i a, __m256i b, int imm);
VPCMPD __mmask8 _mm256_mask_cmp_epi32_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPD __mmask8 _mm256_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__m256i a, __m256i b);
VPCMPD __mmask8 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPUD __mmask8 _mm256_cmp_epu32_mask(__m256i a, __m256i b, int imm);
VPCMPUD __mmask8 _mm256_mask_cmp_epu32_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPUD __mmask8 _mm256_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__m256i a, __m256i b);
VPCMPUD __mmask8 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPD __mmask8 _mm_cmp_epi32_mask(__m128i a, __m128i b, int imm);
VPCMPD __mmask8 _mm_mask_cmp_epi32_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPD __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__m128i a, __m128i b);
VPCMPD __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPUD __mmask8 _mm_cmp_epu32_mask(__m128i a, __m128i b, int imm);
VPCMPUD __mmask8 _mm_mask_cmp_epu32_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPUD __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__m128i a, __m128i b);
VPCMPUD __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__mmask8 k, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

VPCMPQ/VPCMPUQ—Compare Packed Integer Values Into Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 1F /r ib VPCMPQ k1 {k2}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed signed quadword integer values in xmm3/m128/m64bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.256.66.0F3A.W1 1F /r ib VPCMPQ k1 {k2}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed signed quadword integer values in ymm3/m256/m64bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.512.66.0F3A.W1 1F /r ib VPCMPQ k1 {k2}, zmm2, zmm3/m512/m64bcst, imm8	A	V/V	AVX512F OR AVX10.1	Compare packed signed quadword integer values in zmm3/m512/m64bcst and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.128.66.0F3A.W1 1E /r ib VPCMPUQ k1 {k2}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed unsigned quadword integer values in xmm3/m128/m64bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.256.66.0F3A.W1 1E /r ib VPCMPUQ k1 {k2}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed unsigned quadword integer values in ymm3/m256/m64bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.512.66.0F3A.W1 1E /r ib VPCMPUQ k1 {k2}, zmm2, zmm3/m512/m64bcst, imm8	A	V/V	AVX512F OR AVX10.1	Compare packed unsigned quadword integer values in zmm3/m512/m64bcst and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Performs a SIMD compare of the packed integer values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPQ/VPCMPUQ performs a comparison between pairs of signed/unsigned quadword integer values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is a mask register k1. Up to 8/4/2 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-19.

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP := EQ;
    1: OP := LT;
    2: OP := LE;
    3: OP := FALSE;
    4: OP := NEQ;
    5: OP := NLT;
    6: OP := NLE;
    7: OP := TRUE;
ESAC;
```

VPCMPQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+63:i] OP SRC2[63:0];
                ELSE CMP := SRC1[i+63:i] OP SRC2[i+63:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0                ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPCMPUQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+63:i] OP SRC2[63:0];
                ELSE CMP := SRC1[i+63:i] OP SRC2[i+63:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0                ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCMPQ __mmask8 _mm512_cmp_epi64_mask(__m512i a, __m512i b, int imm);
VPCMPQ __mmask8 _mm512_mask_cmp_epi64_mask(__mmask8 k, __m512i a, __m512i b, int imm);
VPCMPQ __mmask8 _mm512_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__m512i a, __m512i b);
VPCMPQ __mmask8 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPUQ __mmask8 _mm512_cmp_epu64_mask(__m512i a, __m512i b, int imm);
VPCMPUQ __mmask8 _mm512_mask_cmp_epu64_mask(__mmask8 k, __m512i a, __m512i b, int imm);
VPCMPUQ __mmask8 _mm512_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__m512i a, __m512i b);
VPCMPUQ __mmask8 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPQ __mmask8 _mm256_cmp_epi64_mask(__m256i a, __m256i b, int imm);
VPCMPQ __mmask8 _mm256_mask_cmp_epi64_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPQ __mmask8 _mm256_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__m256i a, __m256i b);
VPCMPQ __mmask8 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPUQ __mmask8 _mm256_cmp_epu64_mask(__m256i a, __m256i b, int imm);
VPCMPUQ __mmask8 _mm256_mask_cmp_epu64_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPUQ __mmask8 _mm256_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__m256i a, __m256i b);
VPCMPUQ __mmask8 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPQ __mmask8 _mm_cmp_epi64_mask(__m128i a, __m128i b, int imm);
VPCMPQ __mmask8 _mm_mask_cmp_epi64_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPQ __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__m128i a, __m128i b);
VPCMPQ __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPUQ __mmask8 _mm_cmp_epu64_mask(__m128i a, __m128i b, int imm);
VPCMPUQ __mmask8 _mm_mask_cmp_epu64_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPUQ __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__m128i a, __m128i b);
VPCMPUQ __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__mmask8 k, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

VPCMPW/VPCMPUW—Compare Packed Word Values Into Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 3F /r ib VPCMPW k1 {k2}, xmm2, xmm3/m128, imm8	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed word integers in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.256.66.0F3A.W1 3F /r ib VPCMPW k1 {k2}, ymm2, ymm3/m256, imm8	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed word integers in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.512.66.0F3A.W1 3F /r ib VPCMPW k1 {k2}, zmm2, zmm3/m512, imm8	A	V/V	AVX512BW OR AVX10.1	Compare packed signed word integers in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.128.66.0F3A.W1 3E /r ib VPCMPUW k1 {k2}, xmm2, xmm3/m128, imm8	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed unsigned word integers in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.256.66.0F3A.W1 3E /r ib VPCMPUW k1 {k2}, ymm2, ymm3/m256, imm8	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed unsigned word integers in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.512.66.0F3A.W1 3E /r ib VPCMPUW k1 {k2}, zmm2, zmm3/m512, imm8	A	V/V	AVX512BW OR AVX10.1	Compare packed unsigned word integers in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed integer word in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPW performs a comparison between pairs of signed word values.

VPCMPUW performs a comparison between pairs of unsigned word values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand (first operand) is a mask register k1. Up to 32/16/8 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-19.

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP := EQ;
    1: OP := LT;
    2: OP := LE;
    3: OP := FALSE;
    4: OP := NEQ;
    5: OP := NLT;
    6: OP := NLE;
    7: OP := TRUE;
ESAC;
```

VPCMPW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k2[j] OR *no writemask*
        THEN
            ICMP := SRC1[i+15:i] OP SRC2[i+15:i];
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] = 0             ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPCMPUW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k2[j] OR *no writemask*
        THEN
            CMP := SRC1[i+15:i] OP SRC2[i+15:i];
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] = 0             ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCMPW __mmask32 _mm512_cmp_epi16_mask(__m512i a, __m512i b, int cmp);
VPCMPW __mmask32 _mm512_mask_cmp_epi16_mask(__mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPW __mmask16 _mm256_cmp_epi16_mask(__m256i a, __m256i b, int cmp);
VPCMPW __mmask16 _mm256_mask_cmp_epi16_mask(__mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPW __mmask8 _mm_cmp_epi16_mask(__m128i a, __m128i b, int cmp);
VPCMPW __mmask8 _mm_mask_cmp_epi16_mask(__mmask8 m, __m128i a, __m128i b, int cmp);
VPCMPW __mmask32 _mm512_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__m512i a, __m512i b);
VPCMPW __mmask32 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__mmask32 m, __m512i a, __m512i b);
VPCMPW __mmask16 _mm256_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__m256i a, __m256i b);
VPCMPW __mmask16 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__mmask16 m, __m256i a, __m256i b);
VPCMPW __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__m128i a, __m128i b);
VPCMPW __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__mmask8 m, __m128i a, __m128i b);
VPCMPUW __mmask32 _mm512_cmp_epu16_mask(__m512i a, __m512i b, int cmp);
VPCMPUW __mmask32 _mm512_mask_cmp_epu16_mask(__mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPUW __mmask16 _mm256_cmp_epu16_mask(__m256i a, __m256i b, int cmp);
VPCMPUW __mmask16 _mm256_mask_cmp_epu16_mask(__mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPUW __mmask8 _mm_cmp_epu16_mask(__m128i a, __m128i b, int cmp);
VPCMPUW __mmask8 _mm_mask_cmp_epu16_mask(__mmask8 m, __m128i a, __m128i b, int cmp);
VPCMPUW __mmask32 _mm512_cmp[eq|ge|gt|le|lt|neq]_epu16_mask(__m512i a, __m512i b, int cmp);
VPCMPUW __mmask32 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu16_mask(__mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPUW __mmask16 _mm256_cmp[eq|ge|gt|le|lt|neq]_epu16_mask(__m256i a, __m256i b, int cmp);
VPCMPUW __mmask16 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu16_mask(__mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPUW __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epu16_mask(__m128i a, __m128i b, int cmp);
VPCMPUW __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu16_mask(__mmask8 m, __m128i a, __m128i b, int cmp);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, "Type E4 Class Exception Conditions."

VPCOMPRESSB/VCOMPRESSW—Store Sparse Packed Byte/Word Integer Values Into Dense Memory/Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 63 /r VPCOMPRESSB m128{k1}, xmm1	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Compress up to 128 bits of packed byte values from xmm1 to m128 with writemask k1.
EVEX.128.66.0F38.W0 63 /r VPCOMPRESSB xmm1{k1}{z}, xmm2	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Compress up to 128 bits of packed byte values from xmm2 to xmm1 with writemask k1.
EVEX.256.66.0F38.W0 63 /r VPCOMPRESSB m256{k1}, ymm1	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Compress up to 256 bits of packed byte values from ymm1 to m256 with writemask k1.
EVEX.256.66.0F38.W0 63 /r VPCOMPRESSB ymm1{k1}{z}, ymm2	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Compress up to 256 bits of packed byte values from ymm2 to ymm1 with writemask k1.
EVEX.512.66.0F38.W0 63 /r VPCOMPRESSB m512{k1}, zmm1	A	V/V	AVX512_VBMI2 OR AVX10.1	Compress up to 512 bits of packed byte values from zmm1 to m512 with writemask k1.
EVEX.512.66.0F38.W0 63 /r VPCOMPRESSB zmm1{k1}{z}, zmm2	B	V/V	AVX512_VBMI2 OR AVX10.1	Compress up to 512 bits of packed byte values from zmm2 to zmm1 with writemask k1.
EVEX.128.66.0F38.W1 63 /r VPCOMPRESSW m128{k1}, xmm1	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Compress up to 128 bits of packed word values from xmm1 to m128 with writemask k1.
EVEX.128.66.0F38.W1 63 /r VPCOMPRESSW xmm1{k1}{z}, xmm2	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Compress up to 128 bits of packed word values from xmm2 to xmm1 with writemask k1.
EVEX.256.66.0F38.W1 63 /r VPCOMPRESSW m256{k1}, ymm1	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Compress up to 256 bits of packed word values from ymm1 to m256 with writemask k1.
EVEX.256.66.0F38.W1 63 /r VPCOMPRESSW ymm1{k1}{z}, ymm2	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Compress up to 256 bits of packed word values from ymm2 to ymm1 with writemask k1.
EVEX.512.66.0F38.W1 63 /r VPCOMPRESSW m512{k1}, zmm1	A	V/V	AVX512_VBMI2 OR AVX10.1	Compress up to 512 bits of packed word values from zmm1 to m512 with writemask k1.
EVEX.512.66.0F38.W1 63 /r VPCOMPRESSW zmm1{k1}{z}, zmm2	B	V/V	AVX512_VBMI2 OR AVX10.1	Compress up to 512 bits of packed word values from zmm2 to zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
B	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Compress (stores) up to 64 byte values or 32 word values from the source operand (second operand) to the destination operand (first operand), based on the active elements determined by the writemask operand. Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Moves up to 512 bits of packed byte values from the source operand (second operand) to the destination operand (first operand). This instruction is used to store partial contents of a vector register into a byte vector or single memory location using the active elements in operand writemask.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

This instruction supports memory fault suppression.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VPCOMPRESSB store form

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
k := 0
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.byte[k] := SRC.byte[j]
        k := k +1
```

VPCOMPRESSB reg-reg form

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
k := 0
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.byte[k] := SRC.byte[j]
        k := k + 1
IF *merging-masking*:
    *DEST[VL-1:k*8] remains unchanged*
    ELSE DEST[VL-1:k*8] := 0
DEST[MAX_VL-1:VL] := 0
```

VPCOMPRESSW store form

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
k := 0
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.word[k] := SRC.word[j]
        k := k + 1
```

VPCOMPRESSW reg-reg form

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
k := 0
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.word[k] := SRC.word[j]
        k := k + 1
IF *merging-masking*:
    *DEST[VL-1:k*16] remains unchanged*
    ELSE DEST[VL-1:k*16] := 0
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCOMPRESSB __m128i _mm_mask_compress_epi8(__m128i, __mmask16, __m128i);
VPCOMPRESSB __m128i _mm_maskz_compress_epi8(__mmask16, __m128i);
VPCOMPRESSB __m256i _mm256_mask_compress_epi8(__m256i, __mmask32, __m256i);
VPCOMPRESSB __m256i _mm256_maskz_compress_epi8(__mmask32, __m256i);
VPCOMPRESSB __m512i _mm512_mask_compress_epi8(__m512i, __mmask64, __m512i);
VPCOMPRESSB __m512i _mm512_maskz_compress_epi8(__mmask64, __m512i);
VPCOMPRESSB void _mm_mask_compressstoreu_epi8(void*, __mmask16, __m128i);
VPCOMPRESSB void _mm256_mask_compressstoreu_epi8(void*, __mmask32, __m256i);
VPCOMPRESSB void _mm512_mask_compressstoreu_epi8(void*, __mmask64, __m512i);
VPCOMPRESSW __m128i _mm_mask_compress_epi16(__m128i, __mmask8, __m128i);
VPCOMPRESSW __m128i _mm_maskz_compress_epi16(__mmask8, __m128i);
VPCOMPRESSW __m256i _mm256_mask_compress_epi16(__m256i, __mmask16, __m256i);
VPCOMPRESSW __m256i _mm256_maskz_compress_epi16(__mmask16, __m256i);
VPCOMPRESSW __m512i _mm512_mask_compress_epi16(__m512i, __mmask32, __m512i);
VPCOMPRESSW __m512i _mm512_maskz_compress_epi16(__mmask32, __m512i);
VPCOMPRESSW void _mm_mask_compressstoreu_epi16(void*, __mmask8, __m128i);
VPCOMPRESSW void _mm256_mask_compressstoreu_epi16(void*, __mmask16, __m256i);
VPCOMPRESSW void _mm512_mask_compressstoreu_epi16(void*, __mmask32, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, "Type E4 Class Exception Conditions."

VPCOMPRESSD—Store Sparse Packed Doubleword Integer Values Into Dense Memory/Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 8B /r VPCOMPRESSD xmm1/m128 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compress packed doubleword integer values from xmm2 to xmm1/m128 using control mask k1.
EVEX.256.66.0F38.W0 8B /r VPCOMPRESSD ymm1/m256 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compress packed doubleword integer values from ymm2 to ymm1/m256 using control mask k1.
EVEX.512.66.0F38.W0 8B /r VPCOMPRESSD zmm1/m512 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Compress packed doubleword integer values from zmm2 to zmm1/m512 using control mask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Compress (store) up to 16/8/4 doubleword integer values from the source operand (second operand) to the destination operand (first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 16 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VPCOMPRESSD (EVEX encoded versions) store form

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
SIZE := 32
k := 0
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no controlmask*
        THEN
            DEST[k+SIZE-1:k] := SRC[i+31:i]
            k := k + SIZE
    FI;
ENDFOR;
```

VPCOMPRESSD (EVEX encoded versions) reg-reg form

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
SIZE := 32
k := 0
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no controlmask*
        THEN
            DEST[k+SIZE-1:k] := SRC[i+31:i]
            k := k + SIZE
    FI;
ENDFOR
IF *merging-masking*
          THEN *DEST[VL-1:k] remains unchanged*
          ELSE DEST[VL-1:k] := 0
FI
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCOMPRESSD __m512i _mm512_mask_compress_epi32(__m512i s, __mmask16 c, __m512i a);
VPCOMPRESSD __m512i _mm512_maskz_compress_epi32(__mmask16 c, __m512i a);
VPCOMPRESSD void _mm512_mask_compressstoreu_epi32(void * a, __mmask16 c, __m512i s);
VPCOMPRESSD __m256i _mm256_mask_compress_epi32(__m256i s, __mmask8 c, __m256i a);
VPCOMPRESSD __m256i _mm256_maskz_compress_epi32(__mmask8 c, __m256i a);
VPCOMPRESSD void _mm256_mask_compressstoreu_epi32(void * a, __mmask8 c, __m256i s);
VPCOMPRESSD __m128i _mm_mask_compress_epi32(__m128i s, __mmask8 c, __m128i a);
VPCOMPRESSD __m128i _mm_maskz_compress_epi32(__mmask8 c, __m128i a);
VPCOMPRESSD void _mm_mask_compressstoreu_epi32(void * a, __mmask8 c, __m128i s);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

VPCOMPRESSQ—Store Sparse Packed Quadword Integer Values Into Dense Memory/Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 8B /r VPCOMPRESSQ xmm1/m128 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compress packed quadword integer values from xmm2 to xmm1/m128 using control mask k1.
EVEX.256.66.0F38.W1 8B /r VPCOMPRESSQ ymm1/m256 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compress packed quadword integer values from ymm2 to ymm1/m256 using control mask k1.
EVEX.512.66.0F38.W1 8B /r VPCOMPRESSQ zmm1/m512 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Compress packed quadword integer values from zmm2 to zmm1/m512 using control mask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Compress (stores) up to 8/4/2 quadword integer values from the source operand (second operand) to the destination operand (first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 8 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VPCOMPRESSQ (EVEX encoded versions) store form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
SIZE := 64
k := 0
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no controlmask*
        THEN
            DEST[k+SIZE-1:k] := SRC[i+63:i]
            k := k + SIZE
    FI;
ENFOR
```

VPCOMPRESSQ (EVEX encoded versions) reg-reg form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
SIZE := 64
k := 0
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no controlmask*
        THEN
            DEST[k+SIZE-1:k] := SRC[i+63:i]
            k := k + SIZE
    FI;
ENDFOR
IF *merging-masking*
        THEN *DEST[VL-1:k] remains unchanged*
        ELSE DEST[VL-1:k] := 0
FI
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPCOMPRESSQ __m512i _mm512_mask_compress_epi64(__m512i s, __mmask8 c, __m512i a);
VPCOMPRESSQ __m512i _mm512_maskz_compress_epi64( __mmask8 c, __m512i a);
VPCOMPRESSQ void _mm512_mask_compressstoreu_epi64(void * a, __mmask8 c, __m512i s);
VPCOMPRESSQ __m256i _mm256_mask_compress_epi64(__m256i s, __mmask8 c, __m256i a);
VPCOMPRESSQ __m256i _mm256_maskz_compress_epi64( __mmask8 c, __m256i a);
VPCOMPRESSQ void _mm256_mask_compressstoreu_epi64(void * a, __mmask8 c, __m256i s);
VPCOMPRESSQ __m128i _mm_mask_compress_epi64(__m128i s, __mmask8 c, __m128i a);
VPCOMPRESSQ __m128i _mm_maskz_compress_epi64( __mmask8 c, __m128i a);
VPCOMPRESSQ void _mm_mask_compressstoreu_epi64(void * a, __mmask8 c, __m128i s);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

VPCONFLICTD/Q—Detect Conflicts Within a Vector of Packed Dword/Qword Values Into Dense Memory/ Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 C4 /r VPCONFLICTD xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512CD) OR AVX10.1	Detect duplicate double-word values in xmm2/m128/m32bcst using writemask k1.
EVEX.256.66.0F38.W0 C4 /r VPCONFLICTD ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512VL AND AVX512CD) OR AVX10.1	Detect duplicate double-word values in ymm2/m256/m32bcst using writemask k1.
EVEX.512.66.0F38.W0 C4 /r VPCONFLICTD zmm1 {k1}{z}, zmm2/m512/m32bcst	A	V/V	AVX512CD OR AVX10.1	Detect duplicate double-word values in zmm2/m512/m32bcst using writemask k1.
EVEX.128.66.0F38.W1 C4 /r VPCONFLICTQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512CD) OR AVX10.1	Detect duplicate quad-word values in xmm2/m128/m64bcst using writemask k1.
EVEX.256.66.0F38.W1 C4 /r VPCONFLICTQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512CD) OR AVX10.1	Detect duplicate quad-word values in ymm2/m256/m64bcst using writemask k1.
EVEX.512.66.0F38.W1 C4 /r VPCONFLICTQ zmm1 {k1}{z}, zmm2/m512/m64bcst	A	V/V	AVX512CD OR AVX10.1	Detect duplicate quad-word values in zmm2/m512/m64bcst using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Test each dword/qword element of the source operand (the second operand) for equality with all other elements in the source operand closer to the least significant element. Each element's comparison results form a bit vector, which is then zero extended and written to the destination according to the writemask.

EVEX.512 encoded version: The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPCONFLICTD

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j*32
    IF MaskBit(j) OR *no writemask* THEN
        FOR k := 0 TO j-1
            m := k*32
            IF ((SRC[i+31:i] = SRC[m+31:m])) THEN
                DEST[i+k] := 1
            ELSE
                DEST[i+k] := 0
            FI
        ENDFOR
        DEST[i+31:i+j] := 0
    ELSE
        IF *merging-masking* THEN
            *DEST[i+31:i] remains unchanged*
        ELSE
            DEST[i+31:i] := 0
        FI
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPCONFLICTQ

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
     i := j*64
     IF MaskBit(j) OR *no writemask* THEN
        FOR k := 0 TO j-1
            m := k*64
             IF ((SRC[i+63:i] = SRC[m+63:m])) THEN
                DEST[i+k] := 1
             ELSE
                DEST[i+k] := 0
            FI
         ENDFOR
         DEST[i+63:i+j] := 0
    ELSE
        IF *merging-masking* THEN
             *DEST[i+63:i] remains unchanged*
         ELSE
             DEST[i+63:i] := 0
         FI
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCONFLICTD __m512i _mm512_conflict_epi32(__m512i a);
VPCONFLICTD __m512i _mm512_mask_conflict_epi32(__m512i s, __mmask16 m, __m512i a);
VPCONFLICTD __m512i _mm512_maskz_conflict_epi32(__mmask16 m, __m512i a);
VPCONFLICTQ __m512i _mm512_conflict_epi64(__m512i a);
VPCONFLICTQ __m512i _mm512_mask_conflict_epi64(__m512i s, __mmask8 m, __m512i a);
VPCONFLICTQ __m512i _mm512_maskz_conflict_epi64(__mmask8 m, __m512i a);
VPCONFLICTD __m256i _mm256_conflict_epi32(__m256i a);
VPCONFLICTD __m256i _mm256_mask_conflict_epi32(__m256i s, __mmask8 m, __m256i a);
VPCONFLICTD __m256i _mm256_maskz_conflict_epi32(__mmask8 m, __m256i a);
VPCONFLICTQ __m256i _mm256_conflict_epi64(__m256i a);
VPCONFLICTQ __m256i _mm256_mask_conflict_epi64(__m256i s, __mmask8 m, __m256i a);
VPCONFLICTQ __m256i _mm256_maskz_conflict_epi64(__mmask8 m, __m256i a);
VPCONFLICTD __m128i _mm_conflict_epi32(__m128i a);
VPCONFLICTD __m128i _mm_mask_conflict_epi32(__m128i s, __mmask8 m, __m128i a);
VPCONFLICTD __m128i _mm_maskz_conflict_epi32(__mmask8 m, __m128i a);
VPCONFLICTQ __m128i _mm_conflict_epi64(__m128i a);
VPCONFLICTQ __m128i _mm_mask_conflict_epi64(__m128i s, __mmask8 m, __m128i a);
VPCONFLICTQ __m128i _mm_maskz_conflict_epi64(__mmask8 m, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Table 2-52, "Type E4NF Class Exception Conditions."

VPDPB[SU,UU,SS]D[,S]—Multiply and Add Unsigned and Signed Bytes With and Without Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.F2.0F38.W0 50 /r VPDPBSSD xmm1, xmm2, xmm3/m128	A	V/V	AVX_VNNI_INT8	Multiply groups of 4 pairs of signed bytes in xmm3/m128 with corresponding signed bytes of xmm2, summing those products and adding them to the doubleword result in xmm1.
VEX.256.F2.0F38.W0 50 /r VPDPBSSD ymm1, ymm2, ymm3/m256	A	V/V	AVX_VNNI_INT8	Multiply groups of 4 pairs of signed bytes in ymm3/m256 with corresponding signed bytes of ymm2, summing those products and adding them to the doubleword result in ymm1.
VEX.128.F2.0F38.W0 51 /r VPDPBSSDS xmm1, xmm2, xmm3/m128	A	V/V	AVX_VNNI_INT8	Multiply groups of 4 pairs of signed bytes in xmm3/m128 with corresponding signed bytes of xmm2, summing those products and adding them to the doubleword result, with signed saturation in xmm1.
VEX.256.F2.0F38.W0 51 /r VPDPBSSDS ymm1, ymm2, ymm3/m256	A	V/V	AVX_VNNI_INT8	Multiply groups of 4 pairs of signed bytes in ymm3/m256 with corresponding signed bytes of ymm2, summing those products and adding them to the doubleword result, with signed saturation in ymm1.
VEX.128.F3.0F38.W0 50 /r VPDPBSUD xmm1, xmm2, xmm3/m128	A	V/V	AVX_VNNI_INT8	Multiply groups of 4 pairs of unsigned bytes in xmm3/m128 with corresponding signed bytes of xmm2, summing those products and adding them to doubleword result in xmm1.
VEX.256.F3.0F38.W0 50 /r VPDPBSUD ymm1, ymm2, ymm3/m256	A	V/V	AVX_VNNI_INT8	Multiply groups of 4 pairs of unsigned bytes in ymm3/m256 with corresponding signed bytes of ymm2, summing those products and adding them to doubleword result in ymm1.
VEX.128.F3.0F38.W0 51 /r VPDPBSUDS xmm1, xmm2, xmm3/m128	A	V/V	AVX_VNNI_INT8	Multiply groups of 4 pairs of unsigned bytes in xmm3/m128 with corresponding signed bytes of xmm2, summing those products and adding them to doubleword result, with signed saturation in xmm1.
VEX.256.F3.0F38.W0 51 /r VPDPBSUDS ymm1, ymm2, ymm3/m256	A	V/V	AVX_VNNI_INT8	Multiply groups of 4 pairs of unsigned bytes in ymm3/m256 with corresponding signed bytes of ymm2, summing those products and adding them to doubleword result, with signed saturation in ymm1.
VEX.128.NP.0F38.W0 50 /r VPDPBUUD xmm1, xmm2, xmm3/m128	A	V/V	AVX_VNNI_INT8	Multiply groups of 4 pairs of unsigned bytes in xmm3/m128 with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result in xmm1.
VEX.256.NP.0F38.W0 50 /r VPDPBUUD ymm1, ymm2, ymm3/m256	A	V/V	AVX_VNNI_INT8	Multiply groups of 4 pairs of unsigned bytes in ymm3/m256 with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result in ymm1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.NP.0F38.W0 51 /r VPDPBUUDS xmm1, xmm2, xmm3/m128	A	V/V	AVX_VNNI_INT8	Multiply groups of 4 pairs of unsigned bytes in xmm3/m128 with corresponding unsigned bytes of xmm2, summing those products and adding them to the doubleword result, with unsigned saturation in xmm1.
VEX.256.NP.0F38.W0 51 /r VPDPBUUDS ymm1, ymm2, ymm3/m256	A	V/V	AVX_VNNI_INT8	Multiply groups of 4 pairs of unsigned bytes in ymm3/m256 with corresponding unsigned bytes of ymm2, summing those products and adding them to the doubleword result, with unsigned saturation in ymm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies the individual bytes of the first source operand by the corresponding bytes of the second source operand, producing intermediate word results. The word results are then summed and accumulated in the destination dword element size operand.

For unsigned saturation, when an individual result value is beyond the range of an unsigned doubleword (that is, greater than FFFFF_FFFFH), the saturated unsigned doubleword integer value of FFFF_FFFFH is stored in the doubleword destination.

For signed saturation, when an individual result is beyond the range of a signed doubleword integer (that is, greater than 7FFF_FFFFH or less than 8000_0000H), the saturated value of 7FFF_FFFFH or 8000_0000H, respectively, is written to the destination operand.

Operation

VPDPB[SU,UU,SS]D[,S] dest, src1, src2 (VEX encoded version)

```
VL = (128, 256)
KL = VL/32

ORIGDEST := DEST
FOR i := 0 TO KL-1:

    IF *src1 is signed*:
        src1extend := SIGN_EXTEND // SU, SS
    ELSE:
        src1extend := ZERO_EXTEND // UU
    IF *src2 is signed*:
        src2extend := SIGN_EXTEND // SS
    ELSE:
        src2extend := ZERO_EXTEND // UU, SU

    p1word := src1extend(SRC1.byte[4*i+0]) * src2extend(SRC2.byte[4*i+0])
    p2word := src1extend(SRC1.byte[4*i+1]) * src2extend(SRC2.byte[4*i+1])
    p3word := src1extend(SRC1.byte[4*i+2]) * src2extend(SRC2.byte[4*i+2])
    p4word := src1extend(SRC1.byte[4*i+3]) * src2extend(SRC2.byte[4*i+3])

    IF *saturating*:
```

```
        IF *UU instruction version*:
            DEST.dword[i] := UNSIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1word + p2word + p3word + p4word)
        ELSE:
            DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1word + p2word + p3word + p4word)
    ELSE:
        DEST.dword[i] := ORIGDEST.dword[i] + p1word + p2word + p3word + p4word

DEST[MAXVL-1:VL] := 0
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VPDPBSSD __m128i _mm_dpbssd_epi32 (__m128i __W, __m128i __A, __m128i __B);
VPDPBSSD __m256i _mm256_dpbssd_epi32 (__m256i __W, __m256i __A, __m256i __B);
VPDPBSSDS __m128i _mm_dpbssds_epi32 (__m128i __W, __m128i __A, __m128i __B);
VPDPBSSDS __m256i _mm256_dpbssds_epi32 (__m256i __W, __m256i __A, __m256i __B);
VPDPBSUD __m128i _mm_dpbsud_epi32 (__m128i __W, __m128i __A, __m128i __B);
VPDPBSUD __m256i _mm256_dpbsud_epi32 (__m256i __W, __m256i __A, __m256i __B);
VPDPBSUDS __m128i _mm_dpbsuds_epi32 (__m128i __W, __m128i __A, __m128i __B);
VPDPBSUDS __m256i _mm256_dpbsuds_epi32 (__m256i __W, __m256i __A, __m256i __B);
VPDPBUUD __m128i _mm_dpbuud_epi32 (__m128i __W, __m128i __A, __m128i __B);
VPDPBUUD __m256i _mm256_dpbuud_epi32 (__m256i __W, __m256i __A, __m256i __B);
VPDPBUUDS __m128i _mm_dpbuuds_epi32 (__m128i __W, __m128i __A, __m128i __B);
VPDPBUUDS __m256i _mm256_dpbuuds_epi32 (__m256i __W, __m256i __A, __m256i __B);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions.”

VPDPBUSD—Multiply and Add Unsigned and Signed Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 50 /r VPDPBUSD xmm1, xmm2, xmm3/m128	A	V/V	AVX_VNNI	Multiply groups of 4 pairs of signed bytes in xmm3/m128 with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result in xmm1.
VEX.256.66.0F38.W0 50 /r VPDPBUSD ymm1, ymm2, ymm3/m256	A	V/V	AVX_VNNI	Multiply groups of 4 pairs of signed bytes in ymm3/m256 with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result in ymm1.
EVEX.128.66.0F38.W0 50 /r VPDPBUSD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512_VNNI AND AVX512VL) OR AVX10.1	Multiply groups of 4 pairs of signed bytes in xmm3/m128/m32bcst with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result in xmm1 under writemask k1.
EVEX.256.66.0F38.W0 50 /r VPDPBUSD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512_VNNI AND AVX512VL) OR AVX10.1	Multiply groups of 4 pairs of signed bytes in ymm3/m256/m32bcst with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result in ymm1 under writemask k1.
EVEX.512.66.0F38.W0 50 /r VPDPBUSD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512_VNNI OR AVX10.1	Multiply groups of 4 pairs of signed bytes in zmm3/m512/m32bcst with corresponding unsigned bytes of zmm2, summing those products and adding them to doubleword result in zmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies the individual unsigned bytes of the first source operand by the corresponding signed bytes of the second source operand, producing intermediate signed word results. The word results are then summed and accumulated in the destination dword element size operand.

This instruction supports memory fault suppression.

Operation

VPDPBUSD dest, src1, src2 (VEX encoded versions)

```
VL=(128, 256)
KL=VL/32

ORIGDEST := DEST
FOR i := 0 TO KL-1:

    // Extending to 16b
    // src1extend := ZERO_EXTEND
    // src2extend := SIGN_EXTEND

    p1word := src1extend(SRC1.byte[4*i+0]) * src2extend(SRC2.byte[4*i+0])
    p2word := src1extend(SRC1.byte[4*i+1]) * src2extend(SRC2.byte[4*i+1])
    p3word := src1extend(SRC1.byte[4*i+2]) * src2extend(SRC2.byte[4*i+2])
    p4word := src1extend(SRC1.byte[4*i+3]) * src2extend(SRC2.byte[4*i+3])
    DEST.dword[i] := ORIGDEST.dword[i] + p1word + p2word + p3word + p4word

DEST[MAX_VL-1:VL] := 0
```

VPDPBUSD dest, src1, src2 (EVEX encoded versions)

```
(KL,VL)=(4,128), (8,256), (16,512)
ORIGDEST := DEST
FOR i := 0 TO KL-1:
    IF k1[i] or *no writemask*:
        // Byte elements of SRC1 are zero-extended to 16b and
        // byte elements of SRC2 are sign extended to 16b before multiplication.
        IF SRC2 is memory and EVEX.b == 1:
            t := SRC2.dword[0]
        ELSE:
            t := SRC2.dword[i]
        p1word := ZERO_EXTEND(SRC1.byte[4*i]) * SIGN_EXTEND(t.byte[0])
        p2word := ZERO_EXTEND(SRC1.byte[4*i+1]) * SIGN_EXTEND(t.byte[1])
        p3word := ZERO_EXTEND(SRC1.byte[4*i+2]) * SIGN_EXTEND(t.byte[2])
        p4word := ZERO_EXTEND(SRC1.byte[4*i+3]) * SIGN_EXTEND(t.byte[3])
        DEST.dword[i] := ORIGDEST.dword[i] + p1word + p2word + p3word + p4word
    ELSE IF *zeroing*:
        DEST.dword[i] := 0
    ELSE:    // Merge masking, dest element unchanged
        DEST.dword[i] := ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPDPBUSD __m128i _mm_dpbusd_avx_epi32(__m128i, __m128i, __m128i);
VPDPBUSD __m128i _mm_dpbusd_epi32(__m128i, __m128i, __m128i);
VPDPBUSD __m128i _mm_mask_dpbusd_epi32(__m128i, __mmask8, __m128i, __m128i);
VPDPBUSD __m128i _mm_maskz_dpbusd_epi32(__mmask8, __m128i, __m128i, __m128i);
VPDPBUSD __m256i _mm256_dpbusd_avx_epi32(__m256i, __m256i, __m256i);
VPDPBUSD __m256i _mm256_dpbusd_epi32(__m256i, __m256i, __m256i);
VPDPBUSD __m256i _mm256_mask_dpbusd_epi32(__m256i, __mmask8, __m256i, __m256i);
VPDPBUSD __m256i _mm256_maskz_dpbusd_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPBUSD __m512i _mm512_dpbusd_epi32(__m512i, __m512i, __m512i);
VPDPBUSD __m512i _mm512_mask_dpbusd_epi32(__m512i, __mmask16, __m512i, __m512i);
VPDPBUSD __m512i _mm512_maskz_dpbusd_epi32(__mmask16, __m512i, __m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-51, "Type E4 Class Exception Conditions."

VPDPBUSDS—Multiply and Add Unsigned and Signed Bytes With Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 51 /r VPDPBUSDS xmm1, xmm2, xmm3/m128	A	V/V	AVX_VNNI	Multiply groups of 4 pairs signed bytes in xmm3/m128 with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result, with signed saturation in xmm1.
VEX.256.66.0F38.W0 51 /r VPDPBUSDS ymm1, ymm2, ymm3/m256	A	V/V	AVX_VNNI	Multiply groups of 4 pairs signed bytes in ymm3/m256 with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result, with signed saturation in ymm1.
EVEX.128.66.0F38.W0 51 /r VPDPBUSDS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512_VNNI AND AVX512VL) OR AVX10.1	Multiply groups of 4 pairs signed bytes in xmm3/m128/m32bcst with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result, with signed saturation in xmm1, under writemask k1.
EVEX.256.66.0F38.W0 51 /r VPDPBUSDS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512_VNNI AND AVX512VL) OR AVX10.1	Multiply groups of 4 pairs signed bytes in ymm3/m256/m32bcst with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result, with signed saturation in ymm1, under writemask k1.
EVEX.512.66.0F38.W0 51 /r VPDPBUSDS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512_VNNI OR AVX10.1	Multiply groups of 4 pairs signed bytes in zmm3/m512/m32bcst with corresponding unsigned bytes of zmm2, summing those products and adding them to doubleword result, with signed saturation in zmm1, under writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies the individual unsigned bytes of the first source operand by the corresponding signed bytes of the second source operand, producing intermediate signed word results. The word results are then summed and accumulated in the destination dword element size operand. If the intermediate sum overflows a 32b signed number the result is saturated to either 0x7FFF_FFFF for positive numbers of 0x8000_0000 for negative numbers.

This instruction supports memory fault suppression.

Operation

VPDPBUSDS dest, src1, src2 (VEX encoded versions)

```
VL=(128, 256)
KL=VL/32

ORIGDEST := DEST
FOR i := 0 TO KL-1:
    // Extending to 16b
    // src1extend := ZERO_EXTEND
    // src2extend := SIGN_EXTEND

    p1word := src1extend(SRC1.byte[4*i+0]) * src2extend(SRC2.byte[4*i+0])
    p2word := src1extend(SRC1.byte[4*i+1]) * src2extend(SRC2.byte[4*i+1])
    p3word := src1extend(SRC1.byte[4*i+2]) * src2extend(SRC2.byte[4*i+2])
    p4word := src1extend(SRC1.byte[4*i+3]) * src2extend(SRC2.byte[4*i+3])
    DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1word + p2word + p3word + p4word)

DEST[MAX_VL-1:VL] := 0
```

VPDPBUSDS dest, src1, src2 (EVEX encoded versions)

```
(KL,VL)=(4,128), (8,256), (16,512)
ORIGDEST := DEST
FOR i := 0 TO KL-1:
    IF k1[i] or *no writemask*:
        // Byte elements of SRC1 are zero-extended to 16b and
        // byte elements of SRC2 are sign extended to 16b before multiplication.
        IF SRC2 is memory and EVEX.b == 1:
            t := SRC2.dword[0]
        ELSE:
            t := SRC2.dword[i]
        p1word := ZERO_EXTEND(SRC1.byte[4*i]) * SIGN_EXTEND(t.byte[0])
        p2word := ZERO_EXTEND(SRC1.byte[4*i+1]) * SIGN_EXTEND(t.byte[1])
        p3word := ZERO_EXTEND(SRC1.byte[4*i+2]) * SIGN_EXTEND(t.byte[2])
        p4word := ZERO_EXTEND(SRC1.byte[4*i+3]) *SIGN_EXTEND(t.byte[3])
        DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1word + p2word + p3word + p4word)
    ELSE IF *zeroing*:
        DEST.dword[i] := 0
    ELSE:     // Merge masking, dest element unchanged
        DEST.dword[i] := ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPDPBUSDS __m128i _mm_dpbusds_avx_epi32(__m128i, __m128i, __m128i);
VPDPBUSDS __m128i _mm_dpbusds_epi32(__m128i, __m128i, __m128i);
VPDPBUSDS __m128i _mm_mask_dpbusds_epi32(__m128i, __mmask8, __m128i, __m128i);
VPDPBUSDS __m128i _mm_maskz_dpbusds_epi32(__mmask8, __m128i, __m128i, __m128i);
VPDPBUSDS __m256i _mm256_dpbusds_avx_epi32(__m256i, __m256i, __m256i);
VPDPBUSDS __m256i _mm256_dpbusds_epi32(__m256i, __m256i, __m256i);
VPDPBUSDS __m256i _mm256_mask_dpbusds_epi32(__m256i, __mmask8, __m256i, __m256i);
VPDPBUSDS __m256i _mm256_maskz_dpbusds_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPBUSDS __m512i _mm512_dpbusds_epi32(__m512i, __m512i, __m512i);
VPDPBUSDS __m512i _mm512_mask_dpbusds_epi32(__m512i, __mmask16, __m512i, __m512i);
VPDPBUSDS __m512i _mm512_maskz_dpbusds_epi32(__mmask16, __m512i, __m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-51, "Type E4 Class Exception Conditions."

VPDPWSSD—Multiply and Add Signed Word Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 52 /r VPDPWSSD xmm1, xmm2, xmm3/m128	A	V/V	AVX_VNNI	Multiply groups of 2 pairs signed words in xmm3/m128 with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1.
VEX.256.66.0F38.W0 52 /r VPDPWSSD ymm1, ymm2, ymm3/m256	A	V/V	AVX_VNNI	Multiply groups of 2 pairs signed words in ymm3/m256 with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1.
EVEX.128.66.0F38.W0 52 /r VPDPWSSD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512_VNNI AND AVX512VL) OR AVX10.1	Multiply groups of 2 pairs signed words in xmm3/m128/m32bcst with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1, under writemask k1.
EVEX.256.66.0F38.W0 52 /r VPDPWSSD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512_VNNI AND AVX512VL) OR AVX10.1	Multiply groups of 2 pairs signed words in ymm3/m256/m32bcst with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1, under writemask k1.
EVEX.512.66.0F38.W0 52 /r VPDPWSSD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512_VNNI OR AVX10.1	Multiply groups of 2 pairs signed words in zmm3/m512/m32bcst with corresponding signed words of zmm2, summing those products and adding them to doubleword result in zmm1, under writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies the individual signed words of the first source operand by the corresponding signed words of the second source operand, producing intermediate signed, doubleword results. The adjacent doubleword results are then summed and accumulated in the destination operand.

This instruction supports memory fault suppression.

Operation

VPDPWSSD dest, src1, src2 (VEX encoded versions)

```
VL=(128, 256)
KL=VL/32
ORIGDEST := DEST
FOR i := 0 TO KL-1:
    p1dword := SIGN_EXTEND(SRC1.word[2*i+0]) * SIGN_EXTEND(SRC2.word[2*i+0] )
    p2dword := SIGN_EXTEND(SRC1.word[2*i+1]) * SIGN_EXTEND(SRC2.word[2*i+1] )
    DEST.dword[i] := ORIGDEST.dword[i] + p1dword + p2dword
DEST[MAX_VL-1:VL] := 0
```

VPDPWSSD dest, src1, src2 (EVEX encoded versions)

```
(KL,VL)=(4,128), (8,256), (16,512)
ORIGDEST := DEST
FOR i := 0 TO KL-1:
    IF k1[i] or *no writemask*:
        IF SRC2 is memory and EVEX.b == 1:
            t := SRC2.dword[0]
        ELSE:
            t := SRC2.dword[i]
        p1dword := SIGN_EXTEND(SRC1.word[2*i]) * SIGN_EXTEND(t.word[0])
        p2dword := SIGN_EXTEND(SRC1.word[2*i+1]) * SIGN_EXTEND(t.word[1])
        DEST.dword[i] := ORIGDEST.dword[i] + p1dword + p2dword
    ELSE IF *zeroing*:
        DEST.dword[i] := 0
    ELSE:       // Merge masking, dest element unchanged
        DEST.dword[i] := ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPDPWSSD __m128i _mm_dpwssd_avx_epi32(__m128i, __m128i, __m128i);
VPDPWSSD __m128i _mm_dpwssd_epi32(__m128i, __m128i, __m128i);
VPDPWSSD __m128i _mm_mask_dpwssd_epi32(__m128i, __mmask8, __m128i, __m128i);
VPDPWSSD __m128i _mm_maskz_dpwssd_epi32(__mmask8, __m128i, __m128i, __m128i);
VPDPWSSD __m256i _mm256_dpwssd_avx_epi32(__m256i, __m256i, __m256i);
VPDPWSSD __m256i _mm256_dpwssd_epi32(__m256i, __m256i, __m256i);
VPDPWSSD __m256i _mm256_mask_dpwssd_epi32(__m256i, __mmask8, __m256i, __m256i);
VPDPWSSD __m256i _mm256_maskz_dpwssd_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPWSSD __m512i _mm512_dpwssd_epi32(__m512i, __m512i, __m512i);
VPDPWSSD __m512i _mm512_mask_dpwssd_epi32(__m512i, __mmask16, __m512i, __m512i);
VPDPWSSD __m512i _mm512_maskz_dpwssd_epi32(__mmask16, __m512i, __m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

VPDPWSSDS—Multiply and Add Signed Word Integers With Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 53 /r VPDPWSSDS xmm1, xmm2, xmm3/m128	A	V/V	AVX_VNNI	Multiply groups of 2 pairs of signed words in xmm3/m128 with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1, with signed saturation.
VEX.256.66.0F38.W0 53 /r VPDPWSSDS ymm1, ymm2, ymm3/m256	A	V/V	AVX_VNNI	Multiply groups of 2 pairs of signed words in ymm3/m256 with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1, with signed saturation.
EVEX.128.66.0F38.W0 53 /r VPDPWSSDS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512_VNNI AND AVX512VL) OR AVX10.1	Multiply groups of 2 pairs of signed words in xmm3/m128/m32bcst with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1, with signed saturation, under writemask k1.
EVEX.256.66.0F38.W0 53 /r VPDPWSSDS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512_VNNI AND AVX512VL) OR AVX10.1	Multiply groups of 2 pairs of signed words in ymm3/m256/m32bcst with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1, with signed saturation, under writemask k1.
EVEX.512.66.0F38.W0 53 /r VPDPWSSDS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512_VNNI OR AVX10.1	Multiply groups of 2 pairs of signed words in zmm3/m512/m32bcst with corresponding signed words of zmm2, summing those products and adding them to doubleword result in zmm1, with signed saturation, under writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies the individual signed words of the first source operand by the corresponding signed words of the second source operand, producing intermediate signed, doubleword results. The adjacent doubleword results are then summed and accumulated in the destination operand. If the intermediate sum overflows a 32b signed number, the result is saturated to either 0x7FFF_FFFF for positive numbers of 0x8000_0000 for negative numbers.

This instruction supports memory fault suppression.

Operation

VPDPWSSDS dest, src1, src2 (VEX encoded versions)

```
VL=(128, 256)
KL=VL/32
ORIGDEST := DEST
FOR i := 0 TO KL-1:
    p1dword := SIGN_EXTEND(SRC1.word[2*i+0]) * SIGN_EXTEND(SRC2.word[2*i+0])
    p2dword := SIGN_EXTEND(SRC1.word[2*i+1]) * SIGN_EXTEND(SRC2.word[2*i+1])
    DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1dword + p2dword)
DEST[MAX_VL-1:VL] := 0
```

VPDPWSSDS dest, src1, src2 (EVEX encoded versions)

```
(KL,VL)=(4,128), (8,256), (16,512)
ORIGDEST := DEST
FOR i := 0 TO KL-1:
    IF k1[i] or *no writemask*:
        IF SRC2 is memory and EVEX.b == 1:
            t := SRC2.dword[0]
        ELSE:
            t := SRC2.dword[i]
        p1dword := SIGN_EXTEND(SRC1.word[2*i]) * SIGN_EXTEND(t.word[0])
        p2dword := SIGN_EXTEND(SRC1.word[2*i+1]) * SIGN_EXTEND(t.word[1])
        DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1dword + p2dword)
    ELSE IF *zeroing*:
        DEST.dword[i] := 0
    ELSE:     // Merge masking, dest element unchanged
        DEST.dword[i] := ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPDPWSSDS __m128i _mm_dpwssds_avx_epi32(__m128i, __m128i, __m128i);
VPDPWSSDS __m128i _mm_dpwssds_epi32(__m128i, __m128i, __m128i);
VPDPWSSDS __m128i _mm_mask_dpwssd_epi32(__m128i, __mmask8, __m128i, __m128i);
VPDPWSSDS __m128i _mm_maskz_dpwssd_epi32(__mmask8, __m128i, __m128i, __m128i);
VPDPWSSDS __m256i _mm256_dpwssds_avx_epi32(__m256i, __m256i, __m256i);
VPDPWSSDS __m256i _mm256_dpwssd_epi32(__m256i, __m256i, __m256i);
VPDPWSSDS __m256i _mm256_mask_dpwssd_epi32(__m256i, __mmask8, __m256i, __m256i);
VPDPWSSDS __m256i _mm256_maskz_dpwssd_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPWSSDS __m512i _mm512_dpwssd_epi32(__m512i, __m512i, __m512i);
VPDPWSSDS __m512i _mm512_mask_dpwssd_epi32(__m512i, __mmask16, __m512i, __m512i);
VPDPWSSDS __m512i _mm512_maskz_dpwssd_epi32(__mmask16, __m512i, __m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

VPDPW[SU,US,UU]D[,S]—Multiply and Add Unsigned and Signed Words With and Without Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.F3.0F38.W0 D2 /r VPDPWSUD xmm1, xmm2, xmm3/m128	A	V/V	AVX_VNNI_INT16	Multiply groups of 2 pairs of unsigned words in xmm3/m128 with corresponding signed words of xmm2, summing those products and adding them to the doubleword result in xmm1.
VEX.256.F3.0F38.W0 D2 /r VPDPWSUD ymm1, ymm2, ymm3/m256	A	V/V	AVX_VNNI_INT16	Multiply groups of 2 pairs of unsigned words in ymm3/m256 with corresponding signed words of ymm2, summing those products and adding them to the doubleword result in ymm1.
VEX.128.F3.0F38.W0 D3 /r VPDPWSUDS xmm1, xmm2, xmm3/m128	A	V/V	AVX_VNNI_INT16	Multiply groups of 2 pairs of unsigned words in xmm3/m128 with corresponding signed words of xmm2, summing those products and adding them to the doubleword result, with signed saturation in xmm1.
VEX.256.F3.0F38.W0 D3 /r VPDPWSUDS ymm1, ymm2, ymm3/m256	A	V/V	AVX_VNNI_INT16	Multiply groups of 2 pairs of unsigned words in ymm3/m256 with corresponding signed words of ymm2, summing those products and adding them to the doubleword result, with signed saturation in ymm1.
VEX.128.66.0F38.W0 D2 /r VPDPWUSD xmm1, xmm2, xmm3/m128	A	V/V	AVX_VNNI_INT16	Multiply groups of 2 pairs of signed words in xmm3/m128 with corresponding unsigned words of xmm2, summing those products and adding them to doubleword result in xmm1.
VEX.256.66.0F38.W0 D2 /r VPDPWUSD ymm1, ymm2, ymm3/m256	A	V/V	AVX_VNNI_INT16	Multiply groups of 2 pairs of signed words in ymm3/m256 with corresponding unsigned words of ymm2, summing those products and adding them to doubleword result in ymm1.
VEX.128.66.0F38.W0 D3 /r VPDPWUSDS xmm1, xmm2, xmm3/m128	A	V/V	AVX_VNNI_INT16	Multiply groups of 2 pairs of signed words in xmm3/m128 with corresponding unsigned words of xmm2, summing those products and adding them to doubleword result, with signed saturation in xmm1.
VEX.256.66.0F38.W0 D3 /r VPDPWUSDS ymm1, ymm2, ymm3/m256	A	V/V	AVX_VNNI_INT16	Multiply groups of 2 pairs of signed words in ymm3/m256 with corresponding unsigned words of ymm2, summing those products and adding them to doubleword result, with signed saturation in ymm1.
VEX.128.NP.0F38.W0 D2 /r VPDPWUUD xmm1, xmm2, xmm3/m128	A	V/V	AVX_VNNI_INT16	Multiply groups of 2 pairs of unsigned words in xmm3/m128 with corresponding unsigned words of xmm2, summing those products and adding them to doubleword result in xmm1.
VEX.256.NP.0F38.W0 D2 /r VPDPWUUD ymm1, ymm2, ymm3/m256	A	V/V	AVX_VNNI_INT16	Multiply groups of 2 pairs of unsigned words in ymm3/m256 with corresponding unsigned words of ymm2, summing those products and adding them to doubleword result in ymm1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.NP.0F38.W0 D3 /r VPDPWUUDS xmm1, xmm2, xmm3/m128	A	V/V	AVX_VNNI_INT16	Multiply groups of 2 pairs of unsigned words in xmm3/m128 with corresponding unsigned words of xmm2, summing those products and adding them to the doubleword result, with unsigned saturation in xmm1.
VEX.256.NP.0F38.W0 D3 /r VPDPWUUDS ymm1, ymm2, ymm3/m256	A	V/V	AVX_VNNI_INT16	Multiply groups of 2 pairs of unsigned words in ymm3/m256 with corresponding unsigned words of ymm2, summing those products and adding them to the doubleword result, with unsigned saturation in ymm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies the individual words of the first source operand by the corresponding words of the second source operand, producing intermediate dword results. The dword results are then summed and accumulated in the destination dword element size operand.

For unsigned saturation, when an individual result value is beyond the range of an unsigned doubleword (that is, greater than FFFF_FFFFH), the saturated unsigned doubleword integer value of FFFF_FFFFH is stored in the doubleword destination.

For signed saturation, when an individual result is beyond the range of a signed doubleword integer (that is, greater than 7FFF_FFFFH or less than 8000_0000H), the saturated value of 7FFF_FFFFH or 8000_0000H, respectively, is written to the destination operand.

The EVEX version of VPDPWSSD[,S] was previously introduced with AVX512_VNNI. The VEX version of VPDPWSSD[,S] was previously introduced with AVX_VNNI.

Operation

VPDPW[UU,SU,US]D[,S] dest, src1, src2 (VEX encoded version)

```
VL = (128, 256)
KL = VL/32

ORIGDEST := DEST

IF *src1 is signed*:            // SU
    src1extend := SIGN_EXTEND
ELSE:                           // UU, US
    src1extend := ZERO_EXTEND
IF *src2 is signed*:            // US
    src2extend := SIGN_EXTEND
ELSE:                           // UU, SU
    src2extend := ZERO_EXTEND

FOR i := 0 TO KL-1:
    p1dword := src1extend(SRC1.word[2*i+0]) * src2extend(SRC2.word[2*i+0])
    p2dword := src1extend(SRC1.word[2*i+1]) * src2extend(SRC2.word[2*i+1])
    IF *saturating version*:
        IF *UU instruction version*:
```

```
            DEST.dword[i] := UNSIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1dword + p2dword)
        ELSE:
            DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1dword + p2dword)
    ELSE:
        DEST.dword[i] := ORIGDEST.dword[i] + p1dword + p2dword
DEST[MAX_VL-1:VL] := 0
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VPDPWSUD __m128i _mm_dpwsud_epi32 (__m128i __W, __m128i __A, __m128i __B);
VPDPWSUD __m256i _mm256_dpwsud_epi32 (__m256i __W, __m256i __A, __m256i __B);
VPDPWSUDS __m128i _mm_dpwsuds_epi32 (__m128i __W, __m128i __A, __m128i __B);
VPDPWSUDS __m256i _mm256_dpwsuds_epi32 (__m256i __W, __m256i __A, __m256i __B);
VPDPWUSD __m128i _mm_dpwusd_epi32 (__m128i __W, __m128i __A, __m128i __B);
VPDPWUSD __m256i _mm256_dpwusd_epi32 (__m256i __W, __m256i __A, __m256i __B);
VPDPWUSDS __m128i _mm_dpwusds_epi32 (__m128i __W, __m128i __A, __m128i __B);
VPDPWUSDS __m256i _mm256_dpwusds_epi32 (__m256i __W, __m256i __A, __m256i __B);
VPDPWUUD __m128i _mm_dpwuud_epi32 (__m128i __W, __m128i __A, __m128i __B);
VPDPWUUD __m256i _mm256_dpwuud_epi32 (__m256i __W, __m256i __A, __m256i __B);
VPDPWUUDS __m128i _mm_dpwuuds_epi32 (__m128i __W, __m128i __A, __m128i __B);
VPDPWUUDS __m256i _mm256_dpwuuds_epi32 (__m256i __W, __m256i __A, __m256i __B);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions.”

VPERM2F128—Permute Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 06 /r ib VPERM2F128 ymm1, ymm2, ymm3/m256, imm8	RV MI	V/V	AVX	Permute 128-bit floating-point fields in ymm2 and ymm3/mem using controls from imm8 and store result in ymm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Permute 128 bit floating-point-containing fields from the first source operand (second operand) and second source operand (third operand) using bits in the 8-bit immediate and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

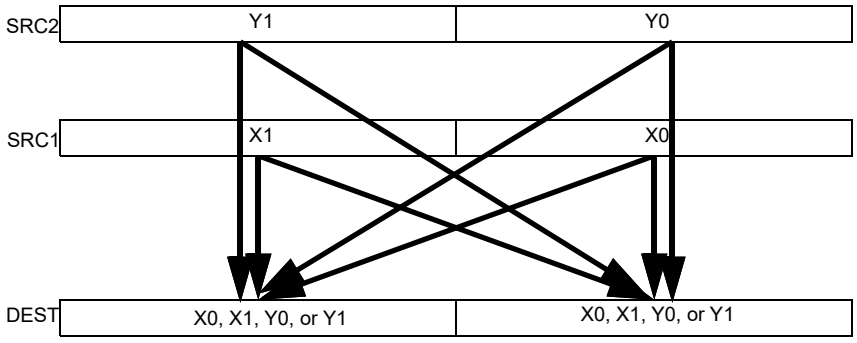


Figure 5-21. VPERM2F128 Operation

Imm8[1:0] select the source for the first destination 128-bit field, imm8[5:4] select the source for the second destination field. If imm8[3] is set, the low 128-bit field is zeroed. If imm8[7] is set, the high 128-bit field is zeroed.

VEX.L must be 1, otherwise the instruction will #UD.

Operation

VPERM2F128

```
CASE IMM8[1:0] of
0: DEST[127:0] := SRC1[127:0]
1: DEST[127:0] := SRC1[255:128]
2: DEST[127:0] := SRC2[127:0]
3: DEST[127:0] := SRC2[255:128]
ESAC

CASE IMM8[5:4] of
0: DEST[255:128] := SRC1[127:0]
1: DEST[255:128] := SRC1[255:128]
2: DEST[255:128] := SRC2[127:0]
3: DEST[255:128] := SRC2[255:128]
ESAC
IF (imm8[3])
DEST[127:0] := 0
FI

IF (imm8[7])
DEST[MAXVL-1:128] := 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERM2F128: __m256 _mm256_permute2f128_ps (__m256 a, __m256 b, int control)

VPERM2F128: __m256d _mm256_permute2f128_pd (__m256d a, __m256d b, int control)

VPERM2F128: __m256i _mm256_permute2f128_si256 (__m256i a, __m256i b, int control)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-23, "Type 6 Class Exception Conditions."

Additionally:

#UD	If VEX.L = 0
	If VEX.W = 1.

VPERM2I128—Permute Integer Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 46 /r ib VPERM2I128 ymm1, ymm2, ymm3/m256, imm8	RVMI	V/V	AVX2	Permute 128-bit integer data in ymm2 and ymm3/mem using controls from imm8 and store result in ymm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Permute 128 bit integer data from the first source operand (second operand) and second source operand (third operand) using bits in the 8-bit immediate and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

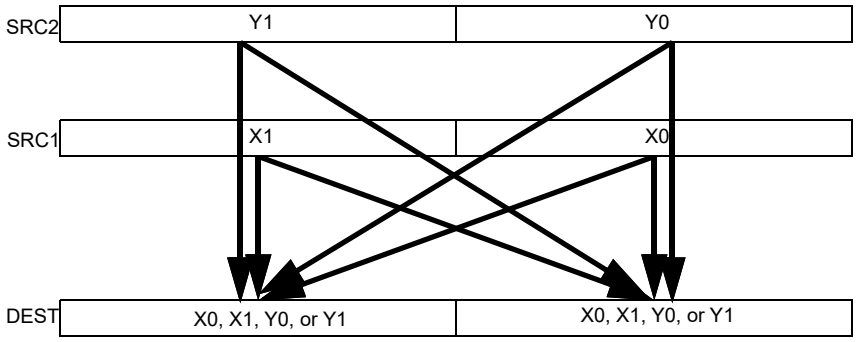


Figure 5-22. VPERM2I128 Operation

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 Table 2-23, "Type 6 Class Exception Conditions."

Additionally:

#UD	If VEX.L = 0, If VEX.W = 1.

VPERMB—Permute Packed Bytes Elements

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 8D /r VPERMB xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	(AVX512VL AND AVX512_VBMI) OR AVX10.1	Permute bytes in xmm3/m128 using byte indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 8D /r VPERMB ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512_VBMI) OR AVX10.1	Permute bytes in ymm3/m256 using byte indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 8D /r VPERMB zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512_VBMI OR AVX10.1	Permute bytes in zmm3/m512 using byte indexes in zmm2 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Copies bytes from the second source operand (the third operand) to the destination operand (the first operand) according to the byte indices in the first source operand (the second operand). Note that this instruction permits a byte in the source operand to be copied to more than one location in the destination operand.

Only the low 6(EVEX.512)/5(EVEX.256)/4(EVEX.128) bits of each byte index is used to select the location of the source byte from the second source operand.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated at byte granularity by the writemask k1.

Operation

VPERMB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
IF VL = 128:
    n := 3;
ELSE IF VL = 256:
    n := 4;
ELSE IF VL = 512:
    n := 5;
FI;
FOR j := 0 TO KL-1:
    id := SRC1[j*8 + n : j*8] ; // location of the source byte
    IF k1[j] OR *no writemask* THEN
        DEST[j*8 + 7: j*8] := SRC2[id*8 +7: id*8];
    ELSE IF zeroing-masking THEN
        DEST[j*8 + 7: j*8] := 0;
    *ELSE
        DEST[j*8 + 7: j*8] remains unchanged*
    FI
ENDFOR
DEST[MAX_VL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMB __m512i _mm512_permutexvar_epi8(__m512i idx, __m512i a);
VPERMB __m512i _mm512_mask_permutexvar_epi8(__m512i s, __mmask64 k, __m512i idx, __m512i a);
VPERMB __m512i _mm512_maskz_permutexvar_epi8(__mmask64 k, __m512i idx, __m512i a);
VPERMB __m256i _mm256_permutexvar_epi8(__m256i idx, __m256i a);
VPERMB __m256i _mm256_mask_permutexvar_epi8(__m256i s, __mmask32 k, __m256i idx, __m256i a);
VPERMB __m256i _mm256_maskz_permutexvar_epi8(__mmask32 k, __m256i idx, __m256i a);
VPERMB __m128i _mm_permutexvar_epi8(__m128i idx, __m128i a);
VPERMB __m128i _mm_mask_permutexvar_epi8(__m128i s, __mmask16 k, __m128i idx, __m128i a);
VPERMB __m128i _mm_maskz_permutexvar_epi8(__mmask16 k, __m128i idx, __m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."

VPERMD/VPERMW—Permute Packed Doubleword/Word Elements

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F38.W0 36 /r VPERMD ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Permute doublewords in ymm3/m256 using indices in ymm2 and store the result in ymm1.
EVEX.256.66.0F38.W0 36 /r VPERMD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute doublewords in ymm3/m256/m32bcst using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 36 /r VPERMD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F OR AVX10.1	Permute doublewords in zmm3/m512/m32bcst using indices in zmm2 and store the result in zmm1 using writemask k1.
EVEX.128.66.0F38.W1 8D /r VPERMW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Permute word integers in xmm3/m128 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 8D /r VPERMW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Permute word integers in ymm3/m256 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 8D /r VPERMW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Permute word integers in zmm3/m512 using indexes in zmm2 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Copies doublewords (or words) from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). Note that this instruction permits a doubleword (word) in the source operand to be copied to more than one location in the destination operand.

VEX.256 encoded VPERMD: The first and second operands are YMM registers, the third operand can be a YMM register or memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded VPERMD: The first and second operands are ZMM/YMM registers, the third operand can be a ZMM/YMM register, a 512/256-bit memory location or a 512/256-bit vector broadcasted from a 32-bit memory location. The elements in the destination are updated using the writemask k1.

VPERMW: first and second operands are ZMM/YMM/XMM registers, the third operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination is updated using the writemask k1.

EVEX.128 encoded versions: Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

Operation

VPERMD (EVEX encoded versions)

```
(KL, VL) = (8, 256), (16, 512)
IF VL = 256 THEN n := 2; FI;
IF VL = 512 THEN n := 3; FI;
FOR j := 0 TO KL-1
    i := j * 32
    id := 32*SRC1[i+n:i]
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := SRC2[31:0];
                ELSE DEST[i+31:i] := SRC2[id+31:id];
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMD (VEX.256 encoded version)

```
DEST[31:0] := (SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
DEST[63:32] := (SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
DEST[95:64] := (SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
DEST[127:96] := (SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
DEST[159:128] := (SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
DEST[191:160] := (SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
DEST[223:192] := (SRC2[255:0] >> (SRC1[194:192] * 32))[31:0];
DEST[255:224] := (SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
DEST[MAXVL-1:256] := 0
```

VPERMW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128 THEN n := 2; FI;
IF VL = 256 THEN n := 3; FI;
IF VL = 512 THEN n := 4; FI;
FOR j := 0 TO KL-1
    i := j * 16
    id := 16*SRC1[i+n:i]
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SRC2[id+15:id]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMD __m512i _mm512_permutexvar_epi32(__m512i idx, __m512i a);
VPERMD __m512i _mm512_mask_permutexvar_epi32(__m512i s, __mmask16 k, __m512i idx, __m512i a);
VPERMD __m512i _mm512_maskz_permutexvar_epi32(__mmask16 k, __m512i idx, __m512i a);
VPERMD __m256i _mm256_permutexvar_epi32(__m256i idx, __m256i a);
VPERMD __m256i _mm256_mask_permutexvar_epi32(__m256i s, __mmask8 k, __m256i idx, __m256i a);
VPERMD __m256i _mm256_maskz_permutexvar_epi32(__mmask8 k, __m256i idx, __m256i a);
VPERMW __m512i _mm512_permutexvar_epi16(__m512i idx, __m512i a);
VPERMW __m512i _mm512_mask_permutexvar_epi16(__m512i s, __mmask32 k, __m512i idx, __m512i a);
VPERMW __m512i _mm512_maskz_permutexvar_epi16(__mmask32 k, __m512i idx, __m512i a);
VPERMW __m256i _mm256_permutexvar_epi16(__m256i idx, __m256i a);
VPERMW __m256i _mm256_mask_permutexvar_epi16(__m256i s, __mmask16 k, __m256i idx, __m256i a);
VPERMW __m256i _mm256_maskz_permutexvar_epi16(__mmask16 k, __m256i idx, __m256i a);
VPERMW __m128i _mm_permutexvar_epi16(__m128i idx, __m128i a);
VPERMW __m128i _mm_mask_permutexvar_epi16(__m128i s, __mmask8 k, __m128i idx, __m128i a);
VPERMW __m128i _mm_maskz_permutexvar_epi16(__mmask8 k, __m128i idx, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded VPERMD, see Table 2-52, "Type E4NF Class Exception Conditions."
EVEX-encoded VPERMW, see Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."
Additionally:

#UD	If VEX.L = 0.
	If EVEX.L'L = 0 for VPERMD.

VPERMI2B—Full Permute of Bytes From Two Tables Overwriting the Index

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 75 /r VPERMI2B xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	(AVX512VL AND AVX512_VBMI) OR AVX10.1	Permute bytes in xmm3/m128 and xmm2 using byte indexes in xmm1 and store the byte results in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 75 /r VPERMI2B ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	(AVX512VL AVX512_VBMI) OR AVX10.1	Permute bytes in ymm3/m256 and ymm2 using byte indexes in ymm1 and store the byte results in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 75 /r VPERMI2B zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512_VBMI OR AVX10.1	Permute bytes in zmm3/m512 and zmm2 using byte indexes in zmm1 and store the byte results in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Permutes byte values in the second operand (the first source operand) and the third operand (the second source operand) using the byte indices in the first operand (the destination operand) to select byte elements from the second or third source operands. The selected byte elements are written to the destination at byte granularity under the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The first operand contains input indices to select elements from the two input tables in the 2nd and 3rd operands. The first operand is also the destination of the result. The third operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. In each index byte, the id bit for table selection is bit 6/5/4, and bits [5:0]/[4:0]/[3:0] selects element within each input table.

Note that these instructions permit a byte value in the source operands to be copied to more than one location in the destination operand. Also, the same tables can be reused in subsequent iterations, but the index elements are overwritten.

Bits (MAX_VL-1:256/128) of the destination are zeroed for VL=256,128.

Operation

VPERMI2B (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
IF VL = 128:
    id := 3;
ELSE IF VL = 256:
    id := 4;
ELSE IF VL = 512:
    id := 5;
FI;
TMP_DEST[VL-1:0] := DEST[VL-1:0];
FOR j := 0 TO KL-1
    off := 8*SRC1[j*8 + id: j*8] ;
    IF k1[j] OR *no writemask*:
        DEST[j*8 + 7: j*8] := TMP_DEST[j*8+id+1]? SRC2[off+7:off] : SRC1[off+7:off];
    ELSE IF *zeroing-masking*
        DEST[j*8 + 7: j*8] := 0;
    *ELSE
        DEST[j*8 + 7: j*8] remains unchanged*
    FI;
ENDFOR
DEST[MAX_VL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMI2B __m512i _mm512_permutex2var_epi8(__m512i a, __m512i idx, __m512i b);
VPERMI2B __m512i _mm512_mask2_permutex2var_epi8(__m512i a, __m512i idx, __mmask64 k, __m512i b);
VPERMI2B __m512i _mm512_maskz_permutex2var_epi8(__mmask64 k, __m512i a, __m512i idx, __m512i b);
VPERMI2B __m256i _mm256_permutex2var_epi8(__m256i a, __m256i idx, __m256i b);
VPERMI2B __m256i _mm256_mask2_permutex2var_epi8(__m256i a, __m256i idx, __mmask32 k, __m256i b);
VPERMI2B __m256i _mm256_maskz_permutex2var_epi8(__mmask32 k, __m256i a, __m256i idx, __m256i b);
VPERMI2B __m128i _mm_permutex2var_epi8(__m128i a, __m128i idx, __m128i b);
VPERMI2B __m128i _mm_mask2_permutex2var_epi8(__m128i a, __m128i idx, __mmask16 k, __m128i b);
VPERMI2B __m128i _mm_maskz_permutex2var_epi8(__mmask16 k, __m128i a, __m128i idx, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."

VPERMI2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting the Index

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 75 /r VPERMI2W xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Permute word integers from two tables in xmm3/m128 and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 75 /r VPERMI2W ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Permute word integers from two tables in ymm3/m256 and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 75 /r VPERMI2W zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512BW OR AVX10.1	Permute word integers from two tables in zmm3/m512 and zmm2 using indexes in zmm1 and store the result in zmm1 using writemask k1.
EVEX.128.66.0F38.W0 76 /r VPERMI2D xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double-words from two tables in xmm3/m128/m32bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 76 /r VPERMI2D ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double-words from two tables in ymm3/m256/m32bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 76 /r VPERMI2D zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F OR AVX10.1	Permute double-words from two tables in zmm3/m512/m32bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.
EVEX.128.66.0F38.W1 76 /r VPERMI2Q xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute quad-words from two tables in xmm3/m128/m64bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 76 /r VPERMI2Q ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute quad-words from two tables in ymm3/m256/m64bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 76 /r VPERMI2Q zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F OR AVX10.1	Permute quad-words from two tables in zmm3/m512/m64bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.
EVEX.128.66.0F38.W0 77 /r VPERMI2PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute single-precision floating-point values from two tables in xmm3/m128/m32bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 77 /r VPERMI2PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute single-precision floating-point values from two tables in ymm3/m256/m32bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 77 /r VPERMI2PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F OR AVX10.1	Permute single-precision floating-point values from two tables in zmm3/m512/m32bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 77 /r VPERMI2PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double precision floating-point values from two tables in xmm3/m128/m64bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 77 /r VPERMI2PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double precision floating-point values from two tables in ymm3/m256/m64bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 77 /r VPERMI2PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F OR AVX10.1	Permute double precision floating-point values from two tables in zmm3/m512/m64bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (r,w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Permutes 16-bit/32-bit/64-bit values in the second operand (the first source operand) and the third operand (the second source operand) using indices in the first operand to select elements from the second and third operands. The selected elements are written to the destination operand (the first operand) according to the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The first operand contains input indices to select elements from the two input tables in the 2nd and 3rd operands. The first operand is also the destination of the result.

D/Q/PS/PD element versions: The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. Broadcast from the low 32/64-bit memory location is performed if EVEX.b and the id bit for table selection are set (selecting table_2).

Dword/PS versions: The id bit for table selection is bit 4/3/2, depending on VL=512, 256, 128. Bits [3:0]/[2:0]/[1:0] of each element in the input index vector select an element within the two source operands, If the id bit is 0, table_1 (the first source) is selected; otherwise the second source operand is selected.

Qword/PD versions: The id bit for table selection is bit 3/2/1, and bits [2:0]/[1:0] /bit 0 selects element within each input table.

Word element versions: The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The id bit for table selection is bit 5/4/3, and bits [4:0]/[3:0]/[2:0] selects element within each input table.

Note that these instructions permit a 16-bit/32-bit/64-bit value in the source operands to be copied to more than one location in the destination operand. Note also that in this case, the same table can be reused for example for a second iteration, while the index elements are overwritten.

Bits (MAXVL-1:256/128) of the destination are zeroed for VL=256,128.

Operation

VPERMI2W (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    id := 2
FI;
IF VL = 256
    id := 3
FI;
IF VL = 512
    id := 4
FI;
TMP_DEST := DEST
FOR j := 0 TO KL-1
    i := j * 16
    off := 16*TMP_DEST[i+id:i]
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+15:i]=TMP_DEST[i+id+1] ? SRC2[off+15:off]
                    : SRC1[off+15:off]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMI2D/VPERMI2PS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    id := 1
FI;
IF VL = 256
    id := 2
FI;
IF VL = 512
    id := 3
FI;
TMP_DEST := DEST
FOR j := 0 TO KL-1
    i := j * 32
    off := 32*TMP_DEST[i+id:i]
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i]  := TMP_DEST[i+id+1] ? SRC2[31:0]
                    : SRC1[off+31:off]
            ELSE
                DEST[i+31:i] := TMP_DEST[i+id+1] ? SRC2[off+31:off]
                    : SRC1[off+31:off]
```

```
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMI2Q/VPERMI2PD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8 512)
IF VL = 128
    id := 0
FI;
IF VL = 256
    id := 1
FI;
IF VL = 512
    id := 2
FI;
TMP_DEST:= DEST
FOR j := 0 TO KL-1
    i := j * 64
    off := 64*TMP_DEST[i+id:i]
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := TMP_DEST[i+id+1] ? SRC2[63:0]
                    : SRC1[off+63:off]
            ELSE
                DEST[i+63:i] := TMP_DEST[i+id+1] ? SRC2[off+63:off]
                    : SRC1[off+63:off]
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMI2D __m512i _mm512_permutex2var_epi32(__m512i a, __m512i idx, __m512i b);
VPERMI2D __m512i _mm512_mask_permutex2var_epi32(__m512i a, __mmask16 k, __m512i idx, __m512i b);
VPERMI2D __m512i _mm512_mask2_permutex2var_epi32(__m512i a, __m512i idx, __mmask16 k, __m512i b);
VPERMI2D __m512i _mm512_maskz_permutex2var_epi32(__mmask16 k, __m512i a, __m512i idx, __m512i b);
VPERMI __m256i _mm256_permutex2var_epi32(__m256i a, __m256i idx, __m256i b);
VPERMI2D __m256i _mm256_mask_permutex2var_epi32(__m256i a, __mmask8 k, __m256i idx, __m256i b);
VPERMI2D __m256i _mm256_mask2_permutex2var_epi32(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERMI2D __m256i _mm256_maskz_permutex2var_epi32(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERMI2D __m128i _mm_permutex2var_epi32(__m128i a, __m128i idx, __m128i b);
VPERMI2D __m128i _mm_mask_permutex2var_epi32(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMI2D __m128i _mm_mask2_permutex2var_epi32(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMI2D __m128i _mm_maskz_permutex2var_epi32(__mmask8 k, __m128i a, __m128i idx, __m128i b);
VPERMI2PD __m512d _mm512_permutex2var_pd(__m512d a, __m512i idx, __m512d b);
VPERMI2PD __m512d _mm512_mask_permutex2var_pd(__m512d a, __mmask8 k, __m512i idx, __m512d b);
VPERMI2PD __m512d _mm512_mask2_permutex2var_pd(__m512d a, __m512i idx, __mmask8 k, __m512d b);
VPERMI2PD __m512d _mm512_maskz_permutex2var_pd(__mmask8 k, __m512d a, __m512i idx, __m512d b);
VPERMI2PD __m256d _mm256_permutex2var_pd(__m256d a, __m256i idx, __m256d b);
VPERMI2PD __m256d _mm256_mask_permutex2var_pd(__m256d a, __mmask8 k, __m256i idx, __m256d b);
VPERMI2PD __m256d _mm256_mask2_permutex2var_pd(__m256d a, __m256i idx, __mmask8 k, __m256d b);
VPERMI2PD __m256d _mm256_maskz_permutex2var_pd(__mmask8 k, __m256d a, __m256i idx, __m256d b);
VPERMI2PD __m128d _mm_permutex2var_pd(__m128d a, __m128i idx, __m128d b);
VPERMI2PD __m128d _mm_mask_permutex2var_pd(__m128d a, __mmask8 k, __m128i idx, __m128d b);
VPERMI2PD __m128d _mm_mask2_permutex2var_pd(__m128d a, __m128i idx, __mmask8 k, __m128d b);
VPERMI2PD __m128d _mm_maskz_permutex2var_pd(__mmask8 k, __m128d a, __m128i idx, __m128d b);
VPERMI2PS __m512 _mm512_permutex2var_ps(__m512 a, __m512i idx, __m512 b);
VPERMI2PS __m512 _mm512_mask_permutex2var_ps(__m512 a, __mmask16 k, __m512i idx, __m512 b);
VPERMI2PS __m512 _mm512_mask2_permutex2var_ps(__m512 a, __m512i idx, __mmask16 k, __m512 b);
VPERMI2PS __m512 _mm512_maskz_permutex2var_ps(__mmask16 k, __m512 a, __m512i idx, __m512 b);
VPERMI2PS __m256 _mm256_permutex2var_ps(__m256 a, __m256i idx, __m256 b);
VPERMI2PS __m256 _mm256_mask_permutex2var_ps(__m256 a, __mmask8 k, __m256i idx, __m256 b);
VPERMI2PS __m256 _mm256_mask2_permutex2var_ps(__m256 a, __m256i idx, __mmask8 k, __m256 b);
VPERMI2PS __m256 _mm256_maskz_permutex2var_ps(__mmask8 k, __m256 a, __m256i idx, __m256 b);
VPERMI2PS __m128 _mm_permutex2var_ps(__m128 a, __m128i idx, __m128 b);
VPERMI2PS __m128 _mm_mask_permutex2var_ps(__m128 a, __mmask8 k, __m128i idx, __m128 b);
VPERMI2PS __m128 _mm_mask2_permutex2var_ps(__m128 a, __m128i idx, __mmask8 k, __m128 b);
VPERMI2PS __m128 _mm_maskz_permutex2var_ps(__mmask8 k, __m128 a, __m128i idx, __m128 b);
VPERMI2Q __m512i _mm512_permutex2var_epi64(__m512i a, __m512i idx, __m512i b);
VPERMI2Q __m512i _mm512_mask_permutex2var_epi64(__m512i a, __mmask8 k, __m512i idx, __m512i b);
VPERMI2Q __m512i _mm512_mask2_permutex2var_epi64(__m512i a, __m512i idx, __mmask8 k, __m512i b);
VPERMI2Q __m512i _mm512_maskz_permutex2var_epi64(__mmask8 k, __m512i a, __m512i idx, __m512i b);
VPERMI2Q __m256i _mm256_permutex2var_epi64(__m256i a, __m256i idx, __m256i b);
VPERMI2Q __m256i _mm256_mask_permutex2var_epi64(__m256i a, __mmask8 k, __m256i idx, __m256i b);
VPERMI2Q __m256i _mm256_mask2_permutex2var_epi64(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERMI2Q __m256i _mm256_maskz_permutex2var_epi64(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERMI2Q __m128i _mm_permutex2var_epi64(__m128i a, __m128i idx, __m128i b);
VPERMI2Q __m128i _mm_mask_permutex2var_epi64(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMI2Q __m128i _mm_mask2_permutex2var_epi64(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMI2Q __m128i _mm_maskz_permutex2var_epi64(__mmask8 k, __m128i a, __m128i idx, __m128i b);

VPERMI2W __m512i _mm512_permutex2var_epi16(__m512i a, __m512i idx, __m512i b);
VPERMI2W __m512i _mm512_mask_permutex2var_epi16(__m512i a, __mmask32 k, __m512i idx, __m512i b);
VPERMI2W __m512i _mm512_mask2_permutex2var_epi16(__m512i a, __m512i idx, __mmask32 k, __m512i b);
VPERMI2W __m512i _mm512_maskz_permutex2var_epi16(__mmask32 k, __m512i a, __m512i idx, __m512i b);
VPERMI2W __m256i _mm256_permutex2var_epi16(__m256i a, __m256i idx, __m256i b);
VPERMI2W __m256i _mm256_mask_permutex2var_epi16(__m256i a, __mmask16 k, __m256i idx, __m256i b);
VPERMI2W __m256i _mm256_mask2_permutex2var_epi16(__m256i a, __m256i idx, __mmask16 k, __m256i b);
VPERMI2W __m256i _mm256_maskz_permutex2var_epi16(__mmask16 k, __m256i a, __m256i idx, __m256i b);
VPERMI2W __m128i _mm_permutex2var_epi16(__m128i a, __m128i idx, __m128i b);
VPERMI2W __m128i _mm_mask_permutex2var_epi16(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMI2W __m128i _mm_mask2_permutex2var_epi16(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMI2W __m128i _mm_maskz_permutex2var_epi16(__mmask8 k, __m128i a, __m128i idx, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VPERMI2D/Q/PS/PD: See Table 2-52, "Type E4NF Class Exception Conditions."
VPERMI2W: See Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."

VPERMILPD—Permute In-Lane of Pairs of Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 0D /r VPERMILPD xmm1, xmm2, xmm3/m128	A	V/V	AVX	Permute double precision floating-point values in xmm2 using controls from xmm3/m128 and store result in xmm1.
VEX.256.66.0F38.W0 0D /r VPERMILPD ymm1, ymm2, ymm3/m256	A	V/V	AVX	Permute double precision floating-point values in ymm2 using controls from ymm3/m256 and store result in ymm1.
EVEX.128.66.0F38.W1 0D /r VPERMILPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double precision floating-point values in xmm2 using control from xmm3/m128/m64bcst and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 0D /r VPERMILPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double precision floating-point values in ymm2 using control from ymm3/m256/m64bcst and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 0D /r VPERMILPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Permute double precision floating-point values in zmm2 using control from zmm3/m512/m64bcst and store the result in zmm1 using writemask k1.
VEX.128.66.0F3A.W0 05 /r ib VPERMILPD xmm1, xmm2/m128, imm8	B	V/V	AVX	Permute double precision floating-point values in xmm2/m128 using controls from imm8.
VEX.256.66.0F3A.W0 05 /r ib VPERMILPD ymm1, ymm2/m256, imm8	B	V/V	AVX	Permute double precision floating-point values in ymm2/m256 using controls from imm8.
EVEX.128.66.0F3A.W1 05 /r ib VPERMILPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double precision floating-point values in xmm2/m128/m64bcst using controls from imm8 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F3A.W1 05 /r ib VPERMILPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double precision floating-point values in ymm2/m256/m64bcst using controls from imm8 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F3A.W1 05 /r ib VPERMILPD zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	D	V/V	AVX512F OR AVX10.1	Permute double precision floating-point values in zmm2/m512/m64bcst using controls from imm8 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

(Variable control version)

Permute pairs of double precision floating-point values in the first source operand (second operand), each using a 1-bit control field residing in the corresponding quadword element of the second source operand (third operand). Permuted results are stored in the destination operand (first operand).

The control bits are located at bit 0 of each quadword element (see Figure 5-24). Each control determines which of the source element in an input pair is selected for the destination element. Each pair of source elements must lie in the same 128-bit region as the destination.

EVEX version: The second source operand (third operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. Permuted results are written to the destination under the writemask.

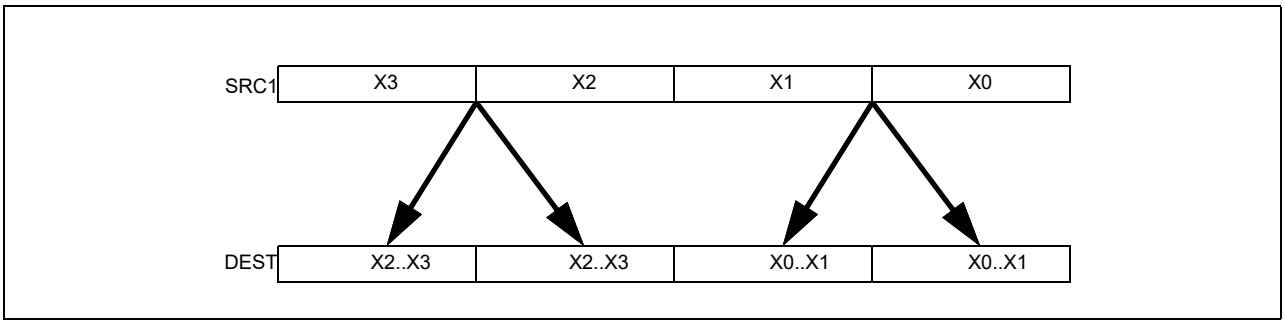


Figure 5-23. VPERMILPD Operation

VEX.256 encoded version: Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

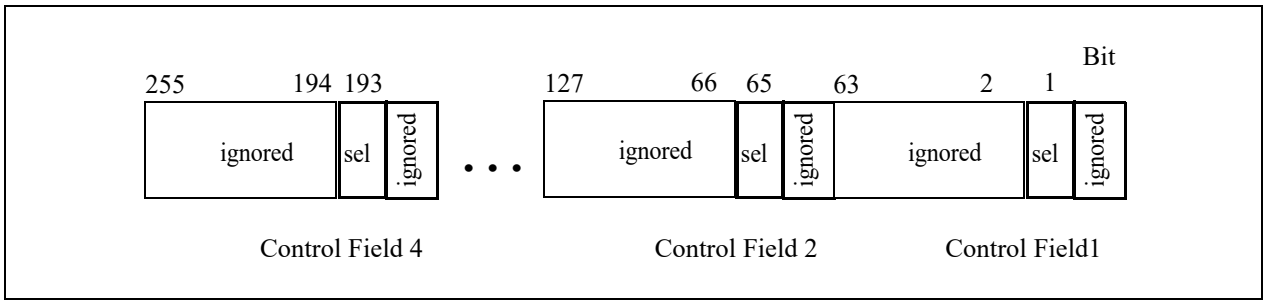


Figure 5-24. VPERMILPD Shuffle Control

Immediate control version: Permute pairs of double precision floating-point values in the first source operand (second operand), each pair using a 1-bit control field in the imm8 byte. Each element in the destination operand (first operand) use a separate control bit of the imm8 byte.

VEX version: The source operand is a YMM/XMM register or a 256/128-bit memory location and the destination operand is a YMM/XMM register. Imm8 byte provides the lower 4/2 bit as permute control fields.

EVEX version: The source operand (second operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. Permuted results are written to the destination under the writemask. Imm8 byte provides the lower 8/4/2 bit as permute control fields.

Note: For the imm8 versions, VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

Operation

VPERMILPD (EVEX immediate versions)

```
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
        THEN TMP_SRC1[i+63:i] := SRC1[63:0];
        ELSE TMP_SRC1[i+63:i] := SRC1[i+63:i];
    FI;
ENDFOR;
IF (imm8[0] = 0) THEN TMP_DEST[63:0] := SRC1[63:0]; FI;
IF (imm8[0] = 1) THEN TMP_DEST[63:0] := TMP_SRC1[127:64]; FI;
IF (imm8[1] = 0) THEN TMP_DEST[127:64] := TMP_SRC1[63:0]; FI;
IF (imm8[1] = 1) THEN TMP_DEST[127:64] := TMP_SRC1[127:64]; FI;
IF VL >= 256
    IF (imm8[2] = 0) THEN TMP_DEST[191:128] := TMP_SRC1[191:128]; FI;
    IF (imm8[2] = 1) THEN TMP_DEST[191:128] := TMP_SRC1[255:192]; FI;
    IF (imm8[3] = 0) THEN TMP_DEST[255:192] := TMP_SRC1[191:128]; FI;
    IF (imm8[3] = 1) THEN TMP_DEST[255:192] := TMP_SRC1[255:192]; FI;
FI;
IF VL >= 512
    IF (imm8[4] = 0) THEN TMP_DEST[319:256] := TMP_SRC1[319:256]; FI;
    IF (imm8[4] = 1) THEN TMP_DEST[319:256] := TMP_SRC1[383:320]; FI;
    IF (imm8[5] = 0) THEN TMP_DEST[383:320] := TMP_SRC1[319:256]; FI;
    IF (imm8[5] = 1) THEN TMP_DEST[383:320] := TMP_SRC1[383:320]; FI;
    IF (imm8[6] = 0) THEN TMP_DEST[447:384] := TMP_SRC1[447:384]; FI;
    IF (imm8[6] = 1) THEN TMP_DEST[447:384] := TMP_SRC1[511:448]; FI;
    IF (imm8[7] = 0) THEN TMP_DEST[511:448] := TMP_SRC1[447:384]; FI;
    IF (imm8[7] = 1) THEN TMP_DEST[511:448] := TMP_SRC1[511:448]; FI;
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMILPD (256-bit immediate version)

```
IF (imm8[0] = 0) THEN DEST[63:0] := SRC1[63:0]
IF (imm8[0] = 1) THEN DEST[63:0] := SRC1[127:64]
IF (imm8[1] = 0) THEN DEST[127:64] := SRC1[63:0]
IF (imm8[1] = 1) THEN DEST[127:64] := SRC1[127:64]
IF (imm8[2] = 0) THEN DEST[191:128] := SRC1[191:128]
IF (imm8[2] = 1) THEN DEST[191:128] := SRC1[255:192]
IF (imm8[3] = 0) THEN DEST[255:192] := SRC1[191:128]
IF (imm8[3] = 1) THEN DEST[255:192] := SRC1[255:192]
DEST[MAXVL-1:256] := 0
```

VPERMILPD (128-bit immediate version)

```
IF (imm8[0] = 0) THEN DEST[63:0] := SRC1[63:0]
IF (imm8[0] = 1) THEN DEST[63:0] := SRC1[127:64]
IF (imm8[1] = 0) THEN DEST[127:64] := SRC1[63:0]
IF (imm8[1] = 1) THEN DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VPERMILPD (EVEX variable versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0];
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i];
    FI;
ENDFOR;

IF (TMP_SRC2[1] = 0) THEN TMP_DEST[63:0] := SRC1[63:0]; FI;
IF (TMP_SRC2[1] = 1) THEN TMP_DEST[63:0] := SRC1[127:64]; FI;
IF (TMP_SRC2[65] = 0) THEN TMP_DEST[127:64] := SRC1[63:0]; FI;
IF (TMP_SRC2[65] = 1) THEN TMP_DEST[127:64] := SRC1[127:64]; FI;
IF VL >= 256
    IF (TMP_SRC2[129] = 0) THEN TMP_DEST[191:128] := SRC1[191:128]; FI;
    IF (TMP_SRC2[129] = 1) THEN TMP_DEST[191:128] := SRC1[255:192]; FI;
    IF (TMP_SRC2[193] = 0) THEN TMP_DEST[255:192] := SRC1[191:128]; FI;
    IF (TMP_SRC2[193] = 1) THEN TMP_DEST[255:192] := SRC1[255:192]; FI;
FI;
IF VL >= 512
    IF (TMP_SRC2[257] = 0) THEN TMP_DEST[319:256] := SRC1[319:256]; FI;
    IF (TMP_SRC2[257] = 1) THEN TMP_DEST[319:256] := SRC1[383:320]; FI;
    IF (TMP_SRC2[321] = 0) THEN TMP_DEST[383:320] := SRC1[319:256]; FI;
    IF (TMP_SRC2[321] = 1) THEN TMP_DEST[383:320] := SRC1[383:320]; FI;
    IF (TMP_SRC2[385] = 0) THEN TMP_DEST[447:384] := SRC1[447:384]; FI;
    IF (TMP_SRC2[385] = 1) THEN TMP_DEST[447:384] := SRC1[511:448]; FI;
    IF (TMP_SRC2[449] = 0) THEN TMP_DEST[511:448] := SRC1[447:384]; FI;
    IF (TMP_SRC2[449] = 1) THEN TMP_DEST[511:448] := SRC1[511:448]; FI;
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
```

```
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMILPD (256-bit variable version)

```
IF (SRC2[1] = 0) THEN DEST[63:0] := SRC1[63:0]
IF (SRC2[1] = 1) THEN DEST[63:0] := SRC1[127:64]
IF (SRC2[65] = 0) THEN DEST[127:64] := SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64] := SRC1[127:64]
IF (SRC2[129] = 0) THEN DEST[191:128] := SRC1[191:128]
IF (SRC2[129] = 1) THEN DEST[191:128] := SRC1[255:192]
IF (SRC2[193] = 0) THEN DEST[255:192] := SRC1[191:128]
IF (SRC2[193] = 1) THEN DEST[255:192] := SRC1[255:192]
DEST[MAXVL-1:256] := 0
```

VPERMILPD (128-bit variable version)

```
IF (SRC2[1] = 0) THEN DEST[63:0] := SRC1[63:0]
IF (SRC2[1] = 1) THEN DEST[63:0] := SRC1[127:64]
IF (SRC2[65] = 0) THEN DEST[127:64] := SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPERMILPD __m512d _mm512_permute_pd( __m512d a, int imm);
VPERMILPD __m512d _mm512_mask_permute_pd(__m512d s, __mmask8 k, __m512d a, int imm);
VPERMILPD __m512d _mm512_maskz_permute_pd( __mmask8 k, __m512d a, int imm);
VPERMILPD __m256d _mm256_mask_permute_pd(__m256d  s, __mmask8 k, __m256d a, int imm);
VPERMILPD __m256d _mm256_maskz_permute_pd( __mmask8 k, __m256d a, int imm);
VPERMILPD __m128d _mm_mask_permute_pd(__m128d s, __mmask8 k, __m128d a, int imm);
VPERMILPD __m128d _mm_maskz_permute_pd( __mmask8 k, __m128d a, int imm);
VPERMILPD __m512d _mm512_permutevar_pd( __m512i i, __m512d a);
VPERMILPD __m512d _mm512_mask_permutevar_pd(__m512d s, __mmask8 k, __m512i i, __m512d a);
VPERMILPD __m512d _mm512_maskz_permutevar_pd( __mmask8 k, __m512i i, __m512d a);
VPERMILPD __m256d _mm256_mask_permutevar_pd(__m256d s, __mmask8 k, __m256d i, __m256d a);
VPERMILPD __m256d _mm256_maskz_permutevar_pd( __mmask8 k, __m256d i, __m256d a);
VPERMILPD __m128d _mm_mask_permutevar_pd(__m128d s, __mmask8 k, __m128d i, __m128d a);
VPERMILPD __m128d _mm_maskz_permutevar_pd( __mmask8 k, __m128d i, __m128d a);
VPERMILPD __m128d _mm_permute_pd (__m128d a, int control)
VPERMILPD __m256d _mm256_permute_pd (__m256d a, int control)
VPERMILPD __m128d _mm_permutevar_pd (__m128d a, __m128i control);
VPERMILPD __m256d _mm256_permutevar_pd (__m256d a, __m256i control);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
Additionally:

#UD	If VEX.W = 1.

EVEX-encoded instruction, see Table 2-52, "Type E4NF Class Exception Conditions."
Additionally:

#UD	If either (E)VEX.vvvv != 1111B and with imm8.

VPERMILPS—Permute In-Lane of Quadruples of Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 0C /r VPERMILPS xmm1, xmm2, xmm3/m128	A	V/V	AVX	Permute single precision floating-point values in xmm2 using controls from xmm3/m128 and store result in xmm1.
VEX.128.66.0F3A.W0 04 /r ib VPERMILPS xmm1, xmm2/m128, imm8	B	V/V	AVX	Permute single precision floating-point values in xmm2/m128 using controls from imm8 and store result in xmm1.
VEX.256.66.0F38.W0 0C /r VPERMILPS ymm1, ymm2, ymm3/m256	A	V/V	AVX	Permute single precision floating-point values in ymm2 using controls from ymm3/m256 and store result in ymm1.
VEX.256.66.0F3A.W0 04 /r ib VPERMILPS ymm1, ymm2/m256, imm8	B	V/V	AVX	Permute single precision floating-point values in ymm2/m256 using controls from imm8 and store result in ymm1.
EVEX.128.66.0F38.W0 0C /r VPERMILPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute single-precision floating-point values xmm2 using control from xmm3/m128/m32bcst and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 0C /r VPERMILPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute single-precision floating-point values ymm2 using control from ymm3/m256/m32bcst and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 0C /r VPERMILPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Permute single-precision floating-point values zmm2 using control from zmm3/m512/m32bcst and store the result in zmm1 using writemask k1.
EVEX.128.66.0F3A.W0 04 /r ib VPERMILPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute single-precision floating-point values xmm2/m128/m32bcst using controls from imm8 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F3A.W0 04 /r ib VPERMILPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute single-precision floating-point values ymm2/m256/m32bcst using controls from imm8 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F3A.W0 04 /r ibVPERMILPS zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	D	V/V	AVX512F OR AVX10.1	Permute single-precision floating-point values zmm2/m512/m32bcst using controls from imm8 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Variable control version:

Permute quadruples of single precision floating-point values in the first source operand (second operand), each quadruplet using a 2-bit control field in the corresponding dword element of the second source operand. Permuted results are stored in the destination operand (first operand).

The 2-bit control fields are located at the low two bits of each dword element (see Figure 5-26). Each control determines which of the source element in an input quadruple is selected for the destination element. Each quadruple of source elements must lie in the same 128-bit region as the destination.

EVEX version: The second source operand (third operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. Permuted results are written to the destination under the writemask.

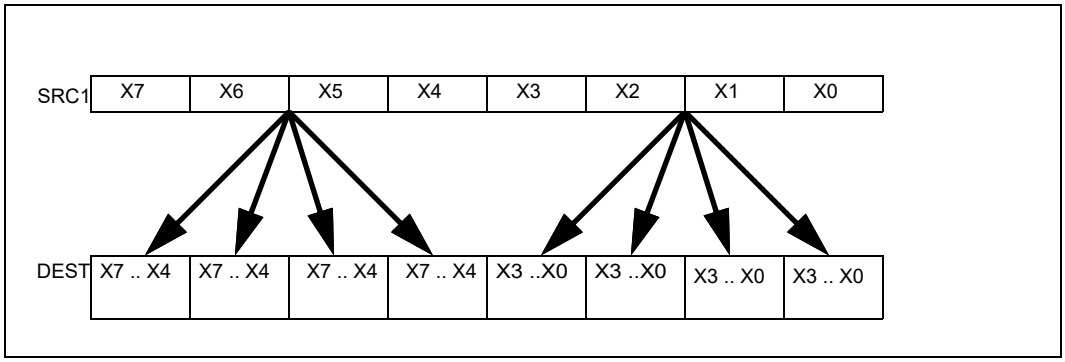


Figure 5-25. VPERMILPS Operation

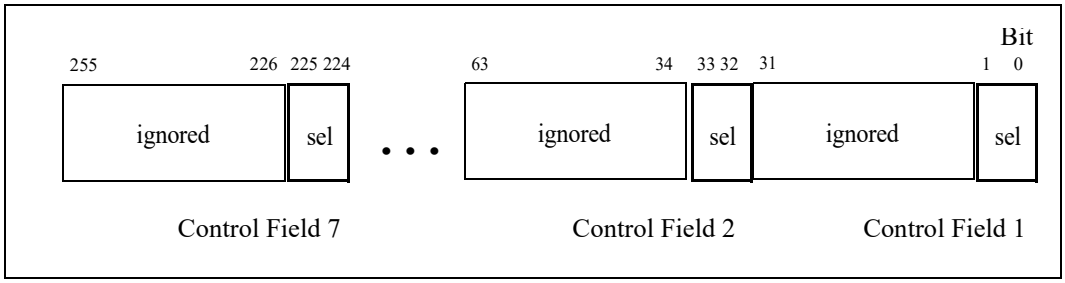


Figure 5-26. VPERMILPS Shuffle Control

(Immediate control version)

Permute quadruples of single precision floating-point values in the first source operand (second operand), each quadruplet using a 2-bit control field in the imm8 byte. Each 128-bit lane in the destination operand (first operand) use the four control fields of the same imm8 byte.

VEX version: The source operand is a YMM/XMM register or a 256/128-bit memory location and the destination operand is a YMM/XMM register.

EVEX version: The source operand (second operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. Permuted results are written to the destination under the writemask.

Note: For the imm8 version, VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

Operation

```
Select4(SRC, control) {
CASE (control[1:0]) OF
    0:    TMP := SRC[31:0];
    1:    TMP := SRC[63:32];
    2:    TMP := SRC[95:64];
    3:    TMP := SRC[127:96];
ESAC;
RETURN TMP
}

VPERMILPS (EVEX immediate versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
        THEN TMP_SRC1[i+31:i] := SRC1[31:0];
        ELSE TMP_SRC1[i+31:i] := SRC1[i+31:i];
    FI;
ENDFOR;

TMP_DEST[31:0] := Select4(TMP_SRC1[127:0], imm8[1:0]);
TMP_DEST[63:32] := Select4(TMP_SRC1[127:0], imm8[3:2]);
TMP_DEST[95:64] := Select4(TMP_SRC1[127:0], imm8[5:4]);
TMP_DEST[127:96] := Select4(TMP_SRC1[127:0], imm8[7:6]); FI;
IF VL >= 256
    TMP_DEST[159:128] := Select4(TMP_SRC1[255:128], imm8[1:0]); FI;
    TMP_DEST[191:160] := Select4(TMP_SRC1[255:128], imm8[3:2]); FI;
    TMP_DEST[223:192] := Select4(TMP_SRC1[255:128], imm8[5:4]); FI;
    TMP_DEST[255:224] := Select4(TMP_SRC1[255:128], imm8[7:6]); FI;
FI;
IF VL >= 512
    TMP_DEST[287:256] := Select4(TMP_SRC1[383:256], imm8[1:0]); FI;
    TMP_DEST[319:288] := Select4(TMP_SRC1[383:256], imm8[3:2]); FI;
    TMP_DEST[351:320] := Select4(TMP_SRC1[383:256], imm8[5:4]); FI;
    TMP_DEST[383:352] := Select4(TMP_SRC1[383:256], imm8[7:6]); FI;
    TMP_DEST[415:384] := Select4(TMP_SRC1[511:384], imm8[1:0]); FI;
    TMP_DEST[447:416] := Select4(TMP_SRC1[511:384], imm8[3:2]); FI;
    TMP_DEST[479:448] := Select4(TMP_SRC1[511:384], imm8[5:4]); FI;
    TMP_DEST[511:480] := Select4(TMP_SRC1[511:384], imm8[7:6]); FI;
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*
                THEN *DEST[i+31:i] remains unchanged*
                ELSE DEST[i+31:i] := 0                    ;zeroing-masking
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMILPS (256-bit immediate version)

```
DEST[31:0] := Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] := Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] := Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] := Select4(SRC1[127:0], imm8[7:6]);
DEST[159:128] := Select4(SRC1[255:128], imm8[1:0]);
DEST[191:160] := Select4(SRC1[255:128], imm8[3:2]);
DEST[223:192] := Select4(SRC1[255:128], imm8[5:4]);
DEST[255:224] := Select4(SRC1[255:128], imm8[7:6]);
```

VPERMILPS (128-bit immediate version)

```
DEST[31:0] := Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] := Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] := Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] := Select4(SRC1[127:0], imm8[7:6]);
DEST[MAXVL-1:128] := 0
```

VPERMILPS (EVEX variable versions)

```
(KL, VL) = (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] := SRC2[31:0];
        ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i];
    FI;
ENDFOR;
TMP_DEST[31:0] := Select4(SRC1[127:0], TMP_SRC2[1:0]);
TMP_DEST[63:32] := Select4(SRC1[127:0], TMP_SRC2[33:32]);
TMP_DEST[95:64] := Select4(SRC1[127:0], TMP_SRC2[65:64]);
TMP_DEST[127:96] := Select4(SRC1[127:0], TMP_SRC2[97:96]);
IF VL >= 256
    TMP_DEST[159:128] := Select4(SRC1[255:128], TMP_SRC2[129:128]);
    TMP_DEST[191:160] := Select4(SRC1[255:128], TMP_SRC2[161:160]);
    TMP_DEST[223:192] := Select4(SRC1[255:128], TMP_SRC2[193:192]);
    TMP_DEST[255:224] := Select4(SRC1[255:128], TMP_SRC2[225:224]);
FI;
IF VL >= 512
    TMP_DEST[287:256] := Select4(SRC1[383:256], TMP_SRC2[257:256]);
    TMP_DEST[319:288] := Select4(SRC1[383:256], TMP_SRC2[289:288]);
    TMP_DEST[351:320] := Select4(SRC1[383:256], TMP_SRC2[321:320]);
    TMP_DEST[383:352] := Select4(SRC1[383:256], TMP_SRC2[353:352]);
    TMP_DEST[415:384] := Select4(SRC1[511:384], TMP_SRC2[385:384]);
    TMP_DEST[447:416] := Select4(SRC1[511:384], TMP_SRC2[417:416]);
    TMP_DEST[479:448] := Select4(SRC1[511:384], TMP_SRC2[449:448]);
    TMP_DEST[511:480] := Select4(SRC1[511:384], TMP_SRC2[481:480]);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*
                THEN *DEST[i+31:i] remains unchanged*
                ELSE DEST[i+31:i] := 0                    ;zeroing-masking
```

```
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMILPS (256-bit variable version)

```
DEST[31:0] := Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] := Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] := Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] := Select4(SRC1[127:0], SRC2[97:96]);
DEST[159:128] := Select4(SRC1[255:128], SRC2[129:128]);
DEST[191:160] := Select4(SRC1[255:128], SRC2[161:160]);
DEST[223:192] := Select4(SRC1[255:128], SRC2[193:192]);
DEST[255:224] := Select4(SRC1[255:128], SRC2[225:224]);
DEST[MAXVL-1:256] := 0
```

VPERMILPS (128-bit variable version)

```
DEST[31:0] := Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] := Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] :=Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] := Select4(SRC1[127:0], SRC2[97:96]);
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPERMILPS __m512 _mm512_permute_ps( __m512 a, int imm);
VPERMILPS __m512 _mm512_mask_permute_ps(__m512 s, __mmask16 k, __m512 a, int imm);
VPERMILPS __m512 _mm512_maskz_permute_ps( __mmask16 k, __m512 a, int imm);
VPERMILPS __m256 _mm256_mask_permute_ps(__m256 s, __mmask8 k, __m256 a, int imm);
VPERMILPS __m256 _mm256_maskz_permute_ps( __mmask8 k, __m256 a, int imm);
VPERMILPS __m128 _mm_mask_permute_ps(__m128 s, __mmask8 k, __m128 a, int imm);
VPERMILPS __m128 _mm_maskz_permute_ps( __mmask8 k, __m128 a, int imm);
VPERMILPS __m512 _mm512_permutevar_ps( __m512i i, __m512 a);
VPERMILPS __m512 _mm512_mask_permutevar_ps(__m512 s, __mmask16 k, __m512i i, __m512 a);
VPERMILPS __m512 _mm512_maskz_permutevar_ps( __mmask16 k, __m512i i, __m512 a);
VPERMILPS __m256 _mm256_mask_permutevar_ps(__m256 s, __mmask8 k, __m256 i, __m256 a);
VPERMILPS __m256 _mm256_maskz_permutevar_ps( __mmask8 k, __m256 i, __m256 a);
VPERMILPS __m128 _mm_mask_permutevar_ps(__m128 s, __mmask8 k, __m128 i, __m128 a);
VPERMILPS __m128 _mm_maskz_permutevar_ps( __mmask8 k, __m128 i, __m128 a);
VPERMILPS __m128 _mm_permute_ps (__m128 a, int control);
VPERMILPS __m256 _mm256_permute_ps (__m256 a, int control);
VPERMILPS __m128 _mm_permutevar_ps (__m128 a, __m128i control);
VPERMILPS __m256 _mm256_permutevar_ps (__m256 a, __m256i control);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
Additionally:

#UD	If VEX.W = 1.

EVEX-encoded instruction, see Table 2-52, "Type E4NF Class Exception Conditions."
Additionally:

#UD	If either (E)VEX.vvvv != 1111B and with imm8.

VPERMPD—Permute Double Precision Floating-Point Elements

Opcode/ Instruction	Op / En	64/32 bit Mode Support	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 indices in imm8 and store the result in ymm1.
EVEX.256.66.0F3A.W1 01 /r ib VPERMPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double precision floating-point elements in ymm2/m256/m64bcst using indexes in imm8 and store the result in ymm1 subject to writemask k1.
EVEX.512.66.0F3A.W1 01 /r ib VPERMPD zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	B	V/V	AVX512F OR AVX10.1	Permute double precision floating-point elements in zmm2/m512/m64bcst using indices in imm8 and store the result in zmm1 subject to writemask k1.
EVEX.256.66.0F38.W1 16 /r VPERMPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double precision floating-point elements in ymm3/m256/m64bcst using indexes in ymm2 and store the result in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W1 16 /r VPERMPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Permute double precision floating-point elements in zmm3/m512/m64bcst using indices in zmm2 and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

The imm8 version: Copies quadword elements of double precision floating-point values from the source operand (the second operand) to the destination operand (the first operand) according to the indices specified by the immediate operand (the third operand). Each two-bit value in the immediate byte selects a qword element in the source operand.

VEX version: The source operand can be a YMM register or a memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

In EVEX.512 encoded version, The elements in the destination are updated using the writemask k1 and the imm8 bits are reused as control bits for the upper 256-bit half when the control bits are coming from immediate. The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location.

The imm8 versions: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The vector control version: Copies quadword elements of double precision floating-point values from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). The first 3 bits of each 64 bit element in the index operand selects which quadword in the second source operand to copy. The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The elements in the destination are updated using the writemask k1.

Note that this instruction permits a qword in the source operand to be copied to multiple locations in the destination operand.

If VPERMPD is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

Operation

VPERMPD (EVEX - imm8 control forms)

```
(KL, VL) = (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC *is memory*)
        THEN TMP_SRC[i+63:i] := SRC[63:0];
        ELSE TMP_SRC[i+63:i] := SRC[i+63:i];
    FI;
ENDFOR;

TMP_DEST[63:0] := (TMP_SRC[256:0] >> (IMM8[1:0] * 64))[63:0];
TMP_DEST[127:64] := (TMP_SRC[256:0] >> (IMM8[3:2] * 64))[63:0];
TMP_DEST[191:128] := (TMP_SRC[256:0] >> (IMM8[5:4] * 64))[63:0];
TMP_DEST[255:192] := (TMP_SRC[256:0] >> (IMM8[7:6] * 64))[63:0];
IF VL >= 512
    TMP_DEST[319:256] := (TMP_SRC[511:256] >> (IMM8[1:0] * 64))[63:0];
    TMP_DEST[383:320] := (TMP_SRC[511:256] >> (IMM8[3:2] * 64))[63:0];
    TMP_DEST[447:384] := (TMP_SRC[511:256] >> (IMM8[5:4] * 64))[63:0];
    TMP_DEST[511:448] := (TMP_SRC[511:256] >> (IMM8[7:6] * 64))[63:0];
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0                       ;zeroing-masking
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMPD (EVEX - vector control forms)

```
(KL, VL) = (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0];
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i];
    FI;
ENDFOR;

IF VL = 256
    TMP_DEST[63:0] := (TMP_SRC2[255:0] >> (SRC1[1:0] * 64))[63:0];
    TMP_DEST[127:64] := (TMP_SRC2[255:0] >> (SRC1[65:64] * 64))[63:0];
    TMP_DEST[191:128] := (TMP_SRC2[255:0] >> (SRC1[129:128] * 64))[63:0];
    TMP_DEST[255:192] := (TMP_SRC2[255:0] >> (SRC1[193:192] * 64))[63:0];
FI;
IF VL = 512
    TMP_DEST[63:0] := (TMP_SRC2[511:0] >> (SRC1[2:0] * 64))[63:0];
    TMP_DEST[127:64] := (TMP_SRC2[511:0] >> (SRC1[66:64] * 64))[63:0];
    TMP_DEST[191:128] := (TMP_SRC2[511:0] >> (SRC1[130:128] * 64))[63:0];
    TMP_DEST[255:192] := (TMP_SRC2[511:0] >> (SRC1[194:192] * 64))[63:0];
    TMP_DEST[319:256] := (TMP_SRC2[511:0] >> (SRC1[258:256] * 64))[63:0];
    TMP_DEST[383:320] := (TMP_SRC2[511:0] >> (SRC1[322:320] * 64))[63:0];
    TMP_DEST[447:384] := (TMP_SRC2[511:0] >> (SRC1[386:384] * 64))[63:0];
    TMP_DEST[511:448] := (TMP_SRC2[511:0] >> (SRC1[450:448] * 64))[63:0];
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0                       ;zeroing-masking
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMPD (VEX.256 encoded version)

```
DEST[63:0] := (SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] := (SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[191:128] := (SRC[255:0] >> (IMM8[5:4] * 64))[63:0];
DEST[255:192] := (SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
DEST[MAXVL-1:256] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMPD __m512d _mm512_permutex_pd(__m512d a, int imm);
VPERMPD __m512d _mm512_mask_permutex_pd(__m512d s, __mmask16 k, __m512d a, int imm);
VPERMPD __m512d _mm512_maskz_permutex_pd(__mmask16 k, __m512d a, int imm);
VPERMPD __m512d _mm512_permutexvar_pd(__m512i i, __m512d a);
VPERMPD __m512d _mm512_mask_permutexvar_pd(__m512d s, __mmask16 k, __m512i i, __m512d a);
VPERMPD __m512d _mm512_maskz_permutexvar_pd(__mmask16 k, __m512i i, __m512d a);
VPERMPD __m256d _mm256_permutex_epi64(__m256d a, int imm);
VPERMPD __m256d _mm256_mask_permutex_epi64(__m256i s, __mmask8 k, __m256d a, int imm);
VPERMPD __m256d _mm256_maskz_permutex_epi64(__mmask8 k, __m256d a, int imm);
VPERMPD __m256d _mm256_permutexvar_epi64(__m256i i, __m256d a);
VPERMPD __m256d _mm256_mask_permutexvar_epi64(__m256i s, __mmask8 k, __m256i i, __m256d a);
VPERMPD __m256d _mm256_maskz_permutexvar_epi64(__mmask8 k, __m256i i, __m256d a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions"; additionally:

#UD	If VEX.L = 0.
	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Table 2-52, "Type E4NF Class Exception Conditions"; additionally:

#UD	If encoded with EVEX.128.
	If EVEX.vvvv != 1111B and with imm8.

VPERMPS—Permute Single Precision Floating-Point Elements

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.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 indices in ymm2 and store the result in ymm1.
EVEX.256.66.0F38.W0 16 /r VPERMPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute single-precision floating-point elements in ymm3/m256/m32bcst using indexes in ymm2 and store the result in ymm1 subject to write mask k1.
EVEX.512.66.0F38.W0 16 /r VPERMPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F OR AVX10.1	Permute single-precision floating-point values in zmm3/m512/m32bcst using indices in zmm2 and store the result in zmm1 subject to write mask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Copies doubleword elements of single precision floating-point values from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). Note that this instruction permits a doubleword in the source operand to be copied to more than one location in the destination operand.

VEX.256 versions: The first and second operands are YMM registers, the third operand can be a YMM register or memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded version: The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The elements in the destination are updated using the writemask k1.

If VPERMPS is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

Operation

VPERMPS (EVEX forms)

```
(KL, VL) (8, 256),= (16, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] := SRC2[31:0];
        ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i];
    FI;
ENDFOR;

IF VL = 256
    TMP_DEST[31:0] := (TMP_SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
    TMP_DEST[63:32] := (TMP_SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
    TMP_DEST[95:64] := (TMP_SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
    TMP_DEST[127:96] := (TMP_SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
    TMP_DEST[159:128] := (TMP_SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
    TMP_DEST[191:160] := (TMP_SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
    TMP_DEST[223:192] := (TMP_SRC2[255:0] >> (SRC1[193:192] * 32))[31:0];
```

```
    TMP_DEST[255:224] := (TMP_SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
FI;
IF VL = 512
    TMP_DEST[31:0] := (TMP_SRC2[511:0] >> (SRC1[3:0] * 32))[31:0];
    TMP_DEST[63:32] := (TMP_SRC2[511:0] >> (SRC1[35:32] * 32))[31:0];
    TMP_DEST[95:64] := (TMP_SRC2[511:0] >> (SRC1[67:64] * 32))[31:0];
    TMP_DEST[127:96] := (TMP_SRC2[511:0] >> (SRC1[99:96] * 32))[31:0];
    TMP_DEST[159:128] := (TMP_SRC2[511:0] >> (SRC1[131:128] * 32))[31:0];
    TMP_DEST[191:160] := (TMP_SRC2[511:0] >> (SRC1[163:160] * 32))[31:0];
    TMP_DEST[223:192] := (TMP_SRC2[511:0] >> (SRC1[195:192] * 32))[31:0];
    TMP_DEST[255:224] := (TMP_SRC2[511:0] >> (SRC1[227:224] * 32))[31:0];
    TMP_DEST[287:256] := (TMP_SRC2[511:0] >> (SRC1[259:256] * 32))[31:0];
    TMP_DEST[319:288] := (TMP_SRC2[511:0] >> (SRC1[291:288] * 32))[31:0];
    TMP_DEST[351:320] := (TMP_SRC2[511:0] >> (SRC1[323:320] * 32))[31:0];
    TMP_DEST[383:352] := (TMP_SRC2[511:0] >> (SRC1[355:352] * 32))[31:0];
    TMP_DEST[415:384] := (TMP_SRC2[511:0] >> (SRC1[387:384] * 32))[31:0];
    TMP_DEST[447:416] := (TMP_SRC2[511:0] >> (SRC1[419:416] * 32))[31:0];
    TMP_DEST[479:448] :=(TMP_SRC2[511:0] >> (SRC1[451:448] * 32))[31:0];
    TMP_DEST[511:480] := (TMP_SRC2[511:0] >> (SRC1[483:480] * 32))[31:0];
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0                       ;zeroing-masking
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMPS (VEX.256 encoded version)

```
DEST[31:0] := (SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
DEST[63:32] := (SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
DEST[95:64] := (SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
DEST[127:96] := (SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
DEST[159:128] := (SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
DEST[191:160] := (SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
DEST[223:192] := (SRC2[255:0] >> (SRC1[194:192] * 32))[31:0];
DEST[255:224] := (SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
DEST[MAXVL-1:256] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMPS __m512 _mm512_permutexvar_ps(__m512i i, __m512 a);
VPERMPS __m512 _mm512_mask_permutexvar_ps(__m512 s, __mmask16 k, __m512i i, __m512 a);
VPERMPS __m512 _mm512_maskz_permutexvar_ps(__mmask16 k, __m512i i, __m512 a);
VPERMPS __m256 _mm256_permutexvar_ps(__m256 i, __m256 a);
VPERMPS __m256 _mm256_mask_permutexvar_ps(__m256 s, __mmask8 k, __m256 i, __m256 a);
VPERMPS __m256 _mm256_maskz_permutexvar_ps(__mmask8 k, __m256 i, __m256 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."

Additionally:

#UD	If VEX.L = 0.

EVEX-encoded instruction, see Table 2-52, "Type E4NF Class Exception Conditions."

VPERMQ—Qwords Element Permutation

Opcode/ Instruction	Op / En	64/32 bit Mode Support	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 indices in imm8 and store the result in ymm1.
EVEX.256.66.0F3A.W1 00 /r ib VPERMQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute qwords in ymm2/m256/m64bcst using indexes in imm8 and store the result in ymm1.
EVEX.512.66.0F3A.W1 00 /r ib VPERMQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	B	V/V	AVX512F OR AVX10.1	Permute qwords in zmm2/m512/m64bcst using indices in imm8 and store the result in zmm1.
EVEX.256.66.0F38.W1 36 /r VPERMQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute qwords in ymm3/m256/m64bcst using indexes in ymm2 and store the result in ymm1.
EVEX.512.66.0F38.W1 36 /r VPERMQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Permute qwords in zmm3/m512/m64bcst using indices in zmm2 and store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

The imm8 version: Copies quadwords from the source operand (the second operand) to the destination operand (the first operand) according to the indices specified by the immediate operand (the third operand). Each two-bit value in the immediate byte selects a qword element in the source operand.

VEX version: The source operand can be a YMM register or a memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

In EVEX.512 encoded version, The elements in the destination are updated using the writemask k1 and the imm8 bits are reused as control bits for the upper 256-bit half when the control bits are coming from immediate. The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location.

Immediate control versions: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The vector control version: Copies quadwords from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). The first 3 bits of each 64 bit element in the index operand selects which quadword in the second source operand to copy. The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The elements in the destination are updated using the writemask k1.

Note that this instruction permits a qword in the source operand to be copied to multiple locations in the destination operand.

If VPERMPQ is encoded with VEX.L= 0 or EVEX.128, an attempt to execute the instruction will cause an #UD exception.

Operation

VPERMQ (EVEX - imm8 control forms)

```
(KL, VL) = (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC *is memory*)
        THEN TMP_SRC[i+63:i] := SRC[63:0];
        ELSE TMP_SRC[i+63:i] := SRC[i+63:i];
    FI;
ENDFOR;
    TMP_DEST[63:0] := (TMP_SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
    TMP_DEST[127:64] := (TMP_SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
    TMP_DEST[191:128] := (TMP_SRC[255:0] >> (IMM8[5:4] * 64))[63:0];
    TMP_DEST[255:192] := (TMP_SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
IF VL >= 512
    TMP_DEST[319:256] := (TMP_SRC[511:256] >> (IMM8[1:0] * 64))[63:0];
    TMP_DEST[383:320] := (TMP_SRC[511:256] >> (IMM8[3:2] * 64))[63:0];
    TMP_DEST[447:384] := (TMP_SRC[511:256] >> (IMM8[5:4] * 64))[63:0];
    TMP_DEST[511:448] := (TMP_SRC[511:256] >> (IMM8[7:6] * 64))[63:0];
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0                       ;zeroing-masking
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMQ (EVEX - vector control forms)

```
(KL, VL) = (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0];
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i];
    FI;
ENDFOR;
IF VL = 256
    TMP_DEST[63:0] := (TMP_SRC2[255:0] >> (SRC1[1:0] * 64))[63:0];
    TMP_DEST[127:64] := (TMP_SRC2[255:0] >> (SRC1[65:64] * 64))[63:0];
    TMP_DEST[191:128] := (TMP_SRC2[255:0] >> (SRC1[129:128] * 64))[63:0];
    TMP_DEST[255:192] := (TMP_SRC2[255:0] >> (SRC1[193:192] * 64))[63:0];
FI;
IF VL = 512
    TMP_DEST[63:0] := (TMP_SRC2[511:0] >> (SRC1[2:0] * 64))[63:0];
    TMP_DEST[127:64] := (TMP_SRC2[511:0] >> (SRC1[66:64] * 64))[63:0];
    TMP_DEST[191:128] := (TMP_SRC2[511:0] >> (SRC1[130:128] * 64))[63:0];
    TMP_DEST[255:192] := (TMP_SRC2[511:0] >> (SRC1[194:192] * 64))[63:0];
    TMP_DEST[319:256] := (TMP_SRC2[511:0] >> (SRC1[258:256] * 64))[63:0];
    TMP_DEST[383:320] := (TMP_SRC2[511:0] >> (SRC1[322:320] * 64))[63:0];
```

```
    TMP_DEST[447:384] := (TMP_SRC2[511:0] >> (SRC1[386:384] * 64))[63:0];
    TMP_DEST[511:448] := (TMP_SRC2[511:0] >> (SRC1[450:448] * 64))[63:0];
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0                       ;zeroing-masking
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMQ (VEX.256 encoded version)

```
DEST[63:0] := (SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] := (SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[191:128] := (SRC[255:0] >> (IMM8[5:4] * 64))[63:0];
DEST[255:192] := (SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
DEST[MAXVL-1:256] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPERMQ __m512i _mm512_permutex_epi64( __m512i a, int imm);
VPERMQ __m512i _mm512_mask_permutex_epi64(__m512i s, __mmask8 k, __m512i a, int imm);
VPERMQ __m512i _mm512_maskz_permutex_epi64( __mmask8 k, __m512i a, int imm);
VPERMQ __m512i _mm512_permutexvar_epi64( __m512i a, __m512i b);
VPERMQ __m512i _mm512_mask_permutexvar_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPERMQ __m512i _mm512_maskz_permutexvar_epi64( __mmask8 k, __m512i a, __m512i b);
VPERMQ __m256i _mm256_permutex_epi64( __m256i a, int imm);
VPERMQ __m256i _mm256_mask_permutex_epi64(__m256i s, __mmask8 k, __m256i a, int imm);
VPERMQ __m256i _mm256_maskz_permutex_epi64( __mmask8 k, __m256i a, int imm);
VPERMQ __m256i _mm256_permutexvar_epi64( __m256i a, __m256i b);
VPERMQ __m256i _mm256_mask_permutexvar_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPERMQ __m256i _mm256_maskz_permutexvar_epi64( __mmask8 k, __m256i a, __m256i b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."

Additionally:

#UD	If VEX.L = 0.
	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Table 2-52, "Type E4NF Class Exception Conditions."

Additionally:

#UD	If encoded with EVEX.128.
	If EVEX.vvvv != 1111B and with imm8.

VPERMT2B—Full Permute of Bytes From Two Tables Overwriting a Table

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 7D /r VPERMT2B xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	(AVX512VL AND AVX512_VBMI) OR AVX10.1	Permute bytes in xmm3/m128 and xmm1 using byte indexes in xmm2 and store the byte results in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 7D /r VPERMT2B ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	(AVX512VL AVX512_VBMI) OR AVX10.1	Permute bytes in ymm3/m256 and ymm1 using byte indexes in ymm2 and store the byte results in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 7D /r VPERMT2B zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512_VBMI OR AVX10.1	Permute bytes in zmm3/m512 and zmm1 using byte indexes in zmm2 and store the byte results in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Permutes byte values from two tables, comprising of the first operand (also the destination operand) and the third operand (the second source operand). The second operand (the first source operand) provides byte indices to select byte results from the two tables. The selected byte elements are written to the destination at byte granularity under the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The second operand contains input indices to select elements from the two input tables in the 1st and 3rd operands. The first operand is also the destination of the result. The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. In each index byte, the id bit for table selection is bit 6/5/4, and bits [5:0]/[4:0]/[3:0] selects element within each input table.

Note that these instructions permit a byte value in the source operands to be copied to more than one location in the destination operand. Also, the second table and the indices can be reused in subsequent iterations, but the first table is overwritten.

Bits (MAX_VL-1:256/128) of the destination are zeroed for VL=256,128.

Operation

VPERMT2B (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
IF VL = 128:
    id := 3;
ELSE IF VL = 256:
    id := 4;
ELSE IF VL = 512:
    id := 5;
FI;
TMP_DEST[VL-1:0] := DEST[VL-1:0];
FOR j := 0 TO KL-1
    off := 8*SRC1[j*8 + id: j*8] ;
    IF k1[j] OR *no writemask*:
        DEST[j*8 + 7: j*8] := SRC1[j*8+id+1]? SRC2[off+7:off] : TMP_DEST[off+7:off];
    ELSE IF *zeroing-masking*
        DEST[j*8 + 7: j*8] := 0;
    *ELSE
        DEST[j*8 + 7: j*8] remains unchanged*
    FI;
ENDFOR
DEST[MAX_VL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPERMT2B __m512i _mm512_permutex2var_epi8(__m512i a, __m512i idx, __m512i b);
VPERMT2B __m512i _mm512_mask_permutex2var_epi8(__m512i a, __mmask64 k, __m512i idx, __m512i b);
VPERMT2B __m512i _mm512_maskz_permutex2var_epi8(__mmask64 k, __m512i a, __m512i idx, __m512i b);
VPERMT2B __m256i _mm256_permutex2var_epi8(__m256i a, __m256i idx, __m256i b);
VPERMT2B __m256i _mm256_mask_permutex2var_epi8(__m256i a, __mmask32 k, __m256i idx, __m256i b);
VPERMT2B __m256i _mm256_maskz_permutex2var_epi8(__mmask32 k, __m256i a, __m256i idx, __m256i b);
VPERMT2B __m128i _mm_permutex2var_epi8(__m128i a, __m128i idx, __m128i b);
VPERMT2B __m128i _mm_mask_permutex2var_epi8(__m128i a, __mmask16 k, __m128i idx, __m128i b);
VPERMT2B __m128i _mm_maskz_permutex2var_epi8(__mmask16 k, __m128i a, __m128i idx, __m128i b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4NF.nb in Table 2-52, “Type E4NF Class Exception Conditions.”

VPERMT2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting One Table

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 7D /r VPERMT2W xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Permute word integers from two tables in xmm3/m128 and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 7D /r VPERMT2W ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Permute word integers from two tables in ymm3/m256 and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 7D /r VPERMT2W zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512BW OR AVX10.1	Permute word integers from two tables in zmm3/m512 and zmm1 using indexes in zmm2 and store the result in zmm1 using writemask k1.
EVEX.128.66.0F38.W0 7E /r VPERMT2D xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double-words from two tables in xmm3/m128/m32bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 7E /r VPERMT2D ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double-words from two tables in ymm3/m256/m32bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 7E /r VPERMT2D zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F OR AVX10.1	Permute double-words from two tables in zmm3/m512/m32bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.
EVEX.128.66.0F38.W1 7E /r VPERMT2Q xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute quad-words from two tables in xmm3/m128/m64bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 7E /r VPERMT2Q ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute quad-words from two tables in ymm3/m256/m64bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 7E /r VPERMT2Q zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F OR AVX10.1	Permute quad-words from two tables in zmm3/m512/m64bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.
EVEX.128.66.0F38.W0 7F /r VPERMT2PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute single-precision floating-point values from two tables in xmm3/m128/m32bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 7F /r VPERMT2PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute single-precision floating-point values from two tables in ymm3/m256/m32bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 7F /r VPERMT2PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F OR AVX10.1	Permute single-precision floating-point values from two tables in zmm3/m512/m32bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 7F /r VPERMT2PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double precision floating-point values from two tables in xmm3/m128/m64bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 7F /r VPERMT2PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double precision floating-point values from two tables in ymm3/m256/m64bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 7F /r VPERMT2PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F OR AVX10.1	Permute double precision floating-point values from two tables in zmm3/m512/m64bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (r,w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Permutes 16-bit/32-bit/64-bit values in the first operand and the third operand (the second source operand) using indices in the second operand (the first source operand) to select elements from the first and third operands. The selected elements are written to the destination operand (the first operand) according to the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The second operand contains input indices to select elements from the two input tables in the 1st and 3rd operands. The first operand is also the destination of the result.

D/Q/PS/PD element versions: The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. Broadcast from the low 32/64-bit memory location is performed if EVEX.b and the id bit for table selection are set (selecting table_2).

Dword/PS versions: The id bit for table selection is bit 4/3/2, depending on VL=512, 256, 128. Bits [3:0]/[2:0]/[1:0] of each element in the input index vector select an element within the two source operands, If the id bit is 0, table_1 (the first source) is selected; otherwise the second source operand is selected.

Qword/PD versions: The id bit for table selection is bit 3/2/1, and bits [2:0]/[1:0] /bit 0 selects element within each input table.

Word element versions: The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The id bit for table selection is bit 5/4/3, and bits [4:0]/[3:0]/[2:0] selects element within each input table.

Note that these instructions permit a 16-bit/32-bit/64-bit value in the source operands to be copied to more than one location in the destination operand. Note also that in this case, the same index can be reused for example for a second iteration, while the table elements being permuted are overwritten.

Bits (MAXVL-1:256/128) of the destination are zeroed for VL=256,128.

Operation

VPERMT2W (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    id := 2
FI;
IF VL = 256
    id := 3
FI;
IF VL = 512
    id := 4
FI;
TMP_DEST := DEST
FOR j := 0 TO KL-1
    i := j * 16
    off := 16*SRC1[i+id:i]
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+15:i]=SRC1[i+id+1] ? SRC2[off+15:off]
                    : TMP_DEST[off+15:off]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMT2D/VPERMT2PS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    id := 1
FI;
IF VL = 256
    id := 2
FI;
IF VL = 512
    id := 3
FI;
TMP_DEST := DEST
FOR j := 0 TO KL-1
    i := j * 32
    off := 32*SRC1[i+id:i]
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] := SRC1[i+id+1] ? SRC2[31:0]
                    : TMP_DEST[off+31:off]
            ELSE
                DEST[i+31:i] := SRC1[i+id+1] ? SRC2[off+31:off]
                    : TMP_DEST[off+31:off]
```

```
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMT2Q/VPERMT2PD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8 512)
IF VL = 128
    id := 0
FI;
IF VL = 256
    id := 1
FI;
IF VL = 512
    id := 2
FI;
TMP_DEST:= DEST
FOR j := 0 TO KL-1
    i := j * 64
    off := 64*SRC1[i+id:i]
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := SRC1[i+id+1] ? SRC2[63:0]
                    : TMP_DEST[off+63:off]
            ELSE
                DEST[i+63:i] := SRC1[i+id+1] ? SRC2[off+63:off]
                    : TMP_DEST[off+63:off]
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMT2D __m512i _mm512_permutex2var_epi32(__m512i a, __m512i idx, __m512i b);
VPERMT2D __m512i _mm512_mask_permutex2var_epi32(__m512i a, __mmask16 k, __m512i idx, __m512i b);
VPERMT2D __m512i _mm512_mask2_permutex2var_epi32(__m512i a, __m512i idx, __mmask16 k, __m512i b);
VPERMT2D __m512i _mm512_maskz_permutex2var_epi32(__mmask16 k, __m512i a, __m512i idx, __m512i b);
VPERMT2D __m256i _mm256_permutex2var_epi32(__m256i a, __m256i idx, __m256i b);
VPERMT2D __m256i _mm256_mask_permutex2var_epi32(__m256i a, __mmask8 k, __m256i idx, __m256i b);

VPERMT2D __m256i _mm256_mask2_permutex2var_epi32(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERMT2D __m256i _mm256_maskz_permutex2var_epi32(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERMT2D __m128i _mm_permutex2var_epi32(__m128i a, __m128i idx, __m128i b);
VPERMT2D __m128i _mm_mask_permutex2var_epi32(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMT2D __m128i _mm_mask2_permutex2var_epi32(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMT2D __m128i _mm_maskz_permutex2var_epi32(__mmask8 k, __m128i a, __m128i idx, __m128i b);
VPERMT2PD __m512d _mm512_permutex2var_pd(__m512d a, __m512i idx, __m512d b);
VPERMT2PD __m512d _mm512_mask_permutex2var_pd(__m512d a, __mmask8 k, __m512i idx, __m512d b);
VPERMT2PD __m512d _mm512_mask2_permutex2var_pd(__m512d a, __m512i idx, __mmask8 k, __m512d b);
VPERMT2PD __m512d _mm512_maskz_permutex2var_pd(__mmask8 k, __m512d a, __m512i idx, __m512d b);
VPERMT2PD __m256d _mm256_permutex2var_pd(__m256d a, __m256i idx, __m256d b);
VPERMT2PD __m256d _mm256_mask_permutex2var_pd(__m256d a, __mmask8 k, __m256i idx, __m256d b);
VPERMT2PD __m256d _mm256_mask2_permutex2var_pd(__m256d a, __m256i idx, __mmask8 k, __m256d b);
VPERMT2PD __m256d _mm256_maskz_permutex2var_pd(__mmask8 k, __m256d a, __m256i idx, __m256d b);
VPERMT2PD __m128d _mm_permutex2var_pd(__m128d a, __m128i idx, __m128d b);
VPERMT2PD __m128d _mm_mask_permutex2var_pd(__m128d a, __mmask8 k, __m128i idx, __m128d b);
VPERMT2PD __m128d _mm_mask2_permutex2var_pd(__m128d a, __m128i idx, __mmask8 k, __m128d b);
VPERMT2PD __m128d _mm_maskz_permutex2var_pd(__mmask8 k, __m128d a, __m128i idx, __m128d b);
VPERMT2PS __m512 _mm512_permutex2var_ps(__m512 a, __m512i idx, __m512 b);
VPERMT2PS __m512 _mm512_mask_permutex2var_ps(__m512 a, __mmask16 k, __m512i idx, __m512 b);
VPERMT2PS __m512 _mm512_mask2_permutex2var_ps(__m512 a, __m512i idx, __mmask16 k, __m512 b);
VPERMT2PS __m512 _mm512_maskz_permutex2var_ps(__mmask16 k, __m512 a, __m512i idx, __m512 b);
VPERMT2PS __m256 _mm256_permutex2var_ps(__m256 a, __m256i idx, __m256 b);
VPERMT2PS __m256 _mm256_mask_permutex2var_ps(__m256 a, __mmask8 k, __m256i idx, __m256 b);
VPERMT2PS __m256 _mm256_mask2_permutex2var_ps(__m256 a, __m256i idx, __mmask8 k, __m256 b);
VPERMT2PS __m256 _mm256_maskz_permutex2var_ps(__mmask8 k, __m256 a, __m256i idx, __m256 b);
VPERMT2PS __m128 _mm_permutex2var_ps(__m128 a, __m128i idx, __m128 b);
VPERMT2PS __m128 _mm_mask_permutex2var_ps(__m128 a, __mmask8 k, __m128i idx, __m128 b);
VPERMT2PS __m128 _mm_mask2_permutex2var_ps(__m128 a, __m128i idx, __mmask8 k, __m128 b);
VPERMT2PS __m128 _mm_maskz_permutex2var_ps(__mmask8 k, __m128 a, __m128i idx, __m128 b);
VPERMT2Q __m512i _mm512_permutex2var_epi64(__m512i a, __m512i idx, __m512i b);
VPERMT2Q __m512i _mm512_mask_permutex2var_epi64(__m512i a, __mmask8 k, __m512i idx, __m512i b);
VPERMT2Q __m512i _mm512_mask2_permutex2var_epi64(__m512i a, __m512i idx, __mmask8 k, __m512i b);
VPERMT2Q __m512i _mm512_maskz_permutex2var_epi64(__mmask8 k, __m512i a, __m512i idx, __m512i b);
VPERMT2Q __m256i _mm256_permutex2var_epi64(__m256i a, __m256i idx, __m256i b);
VPERMT2Q __m256i _mm256_mask_permutex2var_epi64(__m256i a, __mmask8 k, __m256i idx, __m256i b);
VPERMT2Q __m256i _mm256_mask2_permutex2var_epi64(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERMT2Q __m256i _mm256_maskz_permutex2var_epi64(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERMT2Q __m128i _mm_permutex2var_epi64(__m128i a, __m128i idx, __m128i b);
VPERMT2Q __m128i _mm_mask_permutex2var_epi64(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMT2Q __m128i _mm_mask2_permutex2var_epi64(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMT2Q __m128i _mm_maskz_permutex2var_epi64(__mmask8 k, __m128i a, __m128i idx, __m128i b);
VPERMT2W __m512i _mm512_permutex2var_epi16(__m512i a, __m512i idx, __m512i b);
VPERMT2W __m512i _mm512_mask_permutex2var_epi16(__m512i a, __mmask32 k, __m512i idx, __m512i b);
VPERMT2W __m512i _mm512_mask2_permutex2var_epi16(__m512i a, __m512i idx, __mmask32 k, __m512i b);
VPERMT2W __m512i _mm512_maskz_permutex2var_epi16(__mmask32 k, __m512i a, __m512i idx, __m512i b);
VPERMT2W __m256i _mm256_permutex2var_epi16(__m256i a, __m256i idx, __m256i b);
VPERMT2W __m256i _mm256_mask_permutex2var_epi16(__m256i a, __mmask16 k, __m256i idx, __m256i b);
VPERMT2W __m256i _mm256_mask2_permutex2var_epi16(__m256i a, __m256i idx, __mmask16 k, __m256i b);

VPERMT2W __m256i _mm256_maskz_permutex2var_epi16(__mmask16 k, __m256i a, __m256i idx, __m256i b);
VPERMT2W __m128i _mm_permutex2var_epi16(__m128i a, __m128i idx, __m128i b);
VPERMT2W __m128i _mm_mask_permutex2var_epi16(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMT2W __m128i _mm_mask2_permutex2var_epi16(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMT2W __m128i _mm_maskz_permutex2var_epi16(__mmask8 k, __m128i a, __m128i idx, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VPERMT2D/Q/PS/PD: See Table 2-52, "Type E4NF Class Exception Conditions."
VPERMT2W: See Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."

VPEXPANDB/VPEXPANDW—Expand Byte/Word Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 62 /r VPEXPANDB xmm1{k1}{z}, m128	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Expands up to 128 bits of packed byte values from m128 to xmm1 with writemask k1.
EVEX.128.66.0F38.W0 62 /r VPEXPANDB xmm1{k1}{z}, xmm2	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Expands up to 128 bits of packed byte values from xmm2 to xmm1 with writemask k1.
EVEX.256.66.0F38.W0 62 /r VPEXPANDB ymm1{k1}{z}, m256	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Expands up to 256 bits of packed byte values from m256 to ymm1 with writemask k1.
EVEX.256.66.0F38.W0 62 /r VPEXPANDB ymm1{k1}{z}, ymm2	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Expands up to 256 bits of packed byte values from ymm2 to ymm1 with writemask k1.
EVEX.512.66.0F38.W0 62 /r VPEXPANDB zmm1{k1}{z}, m512	A	V/V	AVX512_VBMI2 OR AVX10.1	Expands up to 512 bits of packed byte values from m512 to zmm1 with writemask k1.
EVEX.512.66.0F38.W0 62 /r VPEXPANDB zmm1{k1}{z}, zmm2	B	V/V	AVX512_VBMI2 OR AVX10.1	Expands up to 512 bits of packed byte values from zmm2 to zmm1 with writemask k1.
EVEX.128.66.0F38.W1 62 /r VPEXPANDW xmm1{k1}{z}, m128	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Expands up to 128 bits of packed word values from m128 to xmm1 with writemask k1.
EVEX.128.66.0F38.W1 62 /r VPEXPANDW xmm1{k1}{z}, xmm2	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Expands up to 128 bits of packed word values from xmm2 to xmm1 with writemask k1.
EVEX.256.66.0F38.W1 62 /r VPEXPANDW ymm1{k1}{z}, m256	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Expands up to 256 bits of packed word values from m256 to ymm1 with writemask k1.
EVEX.256.66.0F38.W1 62 /r VPEXPANDW ymm1{k1}{z}, ymm2	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Expands up to 256 bits of packed word values from ymm2 to ymm1 with writemask k1.
EVEX.512.66.0F38.W1 62 /r VPEXPANDW zmm1{k1}{z}, m512	A	V/V	AVX512_VBMI2 OR AVX10.1	Expands up to 512 bits of packed word values from m512 to zmm1 with writemask k1.
EVEX.512.66.0F38.W1 62 /r VPEXPANDW zmm1{k1}{z}, zmm2	B	V/V	AVX512_VBMI2 OR AVX10.1	Expands up to 512 bits of packed byte integer values from zmm2 to zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Expands (loads) up to 64 byte integer values or 32 word integer values from the source operand (memory operand) to the destination operand (register operand), based on the active elements determined by the write-mask operand.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Moves 128, 256 or 512 bits of packed byte integer values from the source operand (memory operand) to the destination operand (register operand). This instruction is used to load from an int8 vector register or memory location

while inserting the data into sparse elements of destination vector register using the active elements pointed out by the operand writemask.

This instruction supports memory fault suppression.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VPEXPANDB

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
k := 0
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.byte[j] := SRC.byte[k];
        k := k + 1
        ELSE:
            IF *merging-masking*:
                *DEST.byte[j] remains unchanged*
                ELSE:                       ; zeroing-masking
                    DEST.byte[j] := 0
DEST[MAX_VL-1:VL] := 0
```

VPEXPANDW

```
(KL, VL) = (8,128), (16,256), (32, 512)
k := 0
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.word[j] := SRC.word[k];
        k := k + 1
        ELSE:
            IF *merging-masking*:
                *DEST.word[j] remains unchanged*
                ELSE:                       ; zeroing-masking
                    DEST.word[j] := 0
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPEXPAND __m128i _mm_mask_expand_epi8(__m128i, __mmask16, __m128i);
VPEXPAND __m128i _mm_maskz_expand_epi8(__mmask16, __m128i);
VPEXPAND __m128i _mm_mask_expandloadu_epi8(__m128i, __mmask16, const void*);
VPEXPAND __m128i _mm_maskz_expandloadu_epi8(__mmask16, const void*);
VPEXPAND __m256i _mm256_mask_expand_epi8(__m256i, __mmask32, __m256i);
VPEXPAND __m256i _mm256_maskz_expand_epi8(__mmask32, __m256i);
VPEXPAND __m256i _mm256_mask_expandloadu_epi8(__m256i, __mmask32, const void*);
VPEXPAND __m256i _mm256_maskz_expandloadu_epi8(__mmask32, const void*);
VPEXPAND __m512i _mm512_mask_expand_epi8(__m512i, __mmask64, __m512i);
VPEXPAND __m512i _mm512_maskz_expand_epi8(__mmask64, __m512i);
VPEXPAND __m512i _mm512_mask_expandloadu_epi8(__m512i, __mmask64, const void*);
VPEXPAND __m512i _mm512_maskz_expandloadu_epi8(__mmask64, const void*);
VPEXPANDW __m128i _mm_mask_expand_epi16(__m128i, __mmask8, __m128i);
VPEXPANDW __m128i _mm_maskz_expand_epi16(__mmask8, __m128i);
VPEXPANDW __m128i _mm_mask_expandloadu_epi16(__m128i, __mmask8, const void*);
VPEXPANDW __m128i _mm_maskz_expandloadu_epi16(__mmask8, const void *);
VPEXPANDW __m256i _mm256_mask_expand_epi16(__m256i, __mmask16, __m256i);
VPEXPANDW __m256i _mm256_maskz_expand_epi16(__mmask16, __m256i);
VPEXPANDW __m256i _mm256_mask_expandloadu_epi16(__m256i, __mmask16, const void*);
VPEXPANDW __m256i _mm256_maskz_expandloadu_epi16(__mmask16, const void*);
VPEXPANDW __m512i _mm512_mask_expand_epi16(__m512i, __mmask32, __m512i);
VPEXPANDW __m512i _mm512_maskz_expand_epi16(__mmask32, __m512i);
VPEXPANDW __m512i _mm512_mask_expandloadu_epi16(__m512i, __mmask32, const void*);
VPEXPANDW __m512i _mm512_maskz_expandloadu_epi16(__mmask32, const void*);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, "Type E4 Class Exception Conditions."

VPEXPANDD—Load Sparse Packed Doubleword Integer Values From Dense Memory/Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 89 /r VPEXPANDD xmm1 {k1}{z}, xmm2/m128	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Expand packed double-word integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F38.W0 89 /r VPEXPANDD ymm1 {k1}{z}, ymm2/m256	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Expand packed double-word integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F38.W0 89 /r VPEXPANDD zmm1 {k1}{z}, zmm2/m512	A	V/V	AVX512F OR AVX10.1	Expand packed double-word integer values from zmm2/m512 to zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Expand (load) up to 16 contiguous doubleword integer values of the input vector in the source operand (the second operand) to sparse elements in the destination operand (the first operand), selected by the writemask k1. The destination operand is a ZMM register, the source operand can be a ZMM register or memory location.

The input vector starts from the lowest element in the source operand. The opmask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VPEXPANDD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
k := 0
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+31:i] := SRC[k+31:k];
            k := k + 32
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPEXPANDD __m512i _mm512_mask_expandloadu_epi32(__m512i s, __mmask16 k, void * a);
VPEXPANDD __m512i _mm512_maskz_expandloadu_epi32(__mmask16 k, void * a);
VPEXPANDD __m512i _mm512_mask_expand_epi32(__m512i s, __mmask16 k, __m512i a);
VPEXPANDD __m512i _mm512_maskz_expand_epi32(__mmask16 k, __m512i a);
VPEXPANDD __m256i _mm256_mask_expandloadu_epi32(__m256i s, __mmask8 k, void * a);
VPEXPANDD __m256i _mm256_maskz_expandloadu_epi32(__mmask8 k, void * a);
VPEXPANDD __m256i _mm256_mask_expand_epi32(__m256i s, __mmask8 k, __m256i a);
VPEXPANDD __m256i _mm256_maskz_expand_epi32(__mmask8 k, __m256i a);
VPEXPANDD __m128i _mm_mask_expandloadu_epi32(__m128i s, __mmask8 k, void * a);
VPEXPANDD __m128i _mm_maskz_expandloadu_epi32(__mmask8 k, void * a);
VPEXPANDD __m128i _mm_mask_expand_epi32(__m128i s, __mmask8 k, __m128i a);
VPEXPANDD __m128i _mm_maskz_expand_epi32(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”
Additionally:

#UD	If EVEX.vvvv != 1111B.

VPEXPANDQ—Load Sparse Packed Quadword Integer Values From Dense Memory/Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 89 /r VPEXPANDQ xmm1 {k1}{z}, xmm2/m128	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Expand packed quad-word integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F38.W1 89 /r VPEXPANDQ ymm1 {k1}{z}, ymm2/m256	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Expand packed quad-word integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F38.W1 89 /r VPEXPANDQ zmm1 {k1}{z}, zmm2/m512	A	V/V	AVX512F OR AVX10.1	Expand packed quad-word integer values from zmm2/m512 to zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Expand (load) up to 8 quadword integer values from the source operand (the second operand) to sparse elements in the destination operand (the first operand), selected by the writemask k1. The destination operand is a ZMM register, the source operand can be a ZMM register or memory location.

The input vector starts from the lowest element in the source operand. The opmask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VPEXPANDQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
k := 0
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+63:i] := SRC[k+63:k];
            k := k + 64
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    THEN DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPEXPANDQ __m512i _mm512_mask_expandloadu_epi64(__m512i s, __mmask8 k, void * a);
VPEXPANDQ __m512i _mm512_maskz_expandloadu_epi64(__mmask8 k, void * a);
VPEXPANDQ __m512i _mm512_mask_expand_epi64(__m512i s, __mmask8 k, __m512i a);
VPEXPANDQ __m512i _mm512_maskz_expand_epi64(__mmask8 k, __m512i a);
VPEXPANDQ __m256i _mm256_mask_expandloadu_epi64(__m256i s, __mmask8 k, void * a);
VPEXPANDQ __m256i _mm256_maskz_expandloadu_epi64(__mmask8 k, void * a);
VPEXPANDQ __m256i _mm256_mask_expand_epi64(__m256i s, __mmask8 k, __m256i a);
VPEXPANDQ __m256i _mm256_maskz_expand_epi64(__mmask8 k, __m256i a);
VPEXPANDQ __m128i _mm_mask_expandloadu_epi64(__m128i s, __mmask8 k, void * a);
VPEXPANDQ __m128i _mm_maskz_expandloadu_epi64(__mmask8 k, void * a);
VPEXPANDQ __m128i _mm_mask_expand_epi64(__m128i s, __mmask8 k, __m128i a);
VPEXPANDQ __m128i _mm_maskz_expand_epi64(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”
Additionally:

#UD	If EVEX.vvvv != 1111B.

VPGATHERDD/VPGATHERDQ—Gather Packed Dword, Packed Qword With Signed Dword Indices

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 90 /vsib VPGATHERDD xmm1 {k1}, vm32x	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.256.66.0F38.W0 90 /vsib VPGATHERDD ymm1 {k1}, vm32y	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.512.66.0F38.W0 90 /vsib VPGATHERDD zmm1 {k1}, vm32z	A	V/V	AVX512F OR AVX10.1	Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.128.66.0F38.W1 90 /vsib VPGATHERDQ xmm1 {k1}, vm32x	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.
EVEX.256.66.0F38.W1 90 /vsib VPGATHERDQ ymm1 {k1}, vm32x	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.
EVEX.512.66.0F38.W1 90 /vsib VPGATHERDQ zmm1 {k1}, vm32y	A	V/V	AVX512F OR AVX10.1	Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	N/A	N/A

Description

A set of 16 or 8 doubleword/quadword memory locations pointed to by base address BASE_ADDR and index vector VINDEX with scale SCALE are gathered. The result is written into vector zmm1. The elements are specified via the VSIB (i.e., the index register is a zmm, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register (zmm1) is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.

- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- These instructions do not accept zeroing-masking since the 0 values in k1 are used to determine completion.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has the same disp8*N and alignment rules as for scalar instructions (Tuple 1).

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1 or 4 byte displacement

VPGATHERDD (EVEX encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j]
        THEN DEST[i+31:i] := MEM[BASE_ADDR +
                        SignExtend(VINDEX[i+31:i]) * SCALE + DISP]
             k1[j] := 0
        ELSE *DEST[i+31:i] := remains unchanged*            ; Only merging masking is allowed
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL] := 0
```

VPGATHERDQ (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j]
        THEN DEST[i+63:i] :=
             MEM[BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP]
             k1[j] := 0
        ELSE *DEST[i+63:i] := remains unchanged*            ; Only merging masking is allowed
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPGATHERDD __m512i _mm512_i32gather_epi32(__m512i vdx, void * base, int scale);

VPGATHERDD __m512i _mm512_mask_i32gather_epi32(__m512i s, __mmask16 k, __m512i vdx, void * base, int scale);
VPGATHERDD __m256i _mm256_mmask_i32gather_epi32(__m256i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERDD __m128i _mm_mmask_i32gather_epi32(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERDQ __m512i _mm512_i32logather_epi64(__m256i vdx, void * base, int scale);
VPGATHERDQ __m512i _mm512_mask_i32logather_epi64(__m512i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERDQ __m256i _mm256_mmask_i32logather_epi64(__m256i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERDQ __m128i _mm_mmask_i32gather_epi64(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, "Type E12 Class Exception Conditions."

VPGATHERDD/VPGATHERQD—Gather Packed Dword Values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 90 /r VPGATHERDD xmm1, vm32x, xmm2	RMV	V/V	AVX2	Using dword indices specified in vm32x, gather dword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.
VEX.128.66.0F38.W0 91 /r VPGATHERQD xmm1, vm64x, xmm2	RMV	V/V	AVX2	Using qword indices specified in vm64x, gather dword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.
VEX.256.66.0F38.W0 90 /r VPGATHERDD ymm1, vm32y, ymm2	RMV	V/V	AVX2	Using dword indices specified in vm32y, gather dword from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.
VEX.256.66.0F38.W0 91 /r VPGATHERQD xmm1, vm64y, xmm2	RMV	V/V	AVX2	Using qword indices specified in vm64y, gather dword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	N/A

Description

The instruction conditionally loads up to 4 or 8 dword values from memory addresses specified by the memory operand (the second operand) and using dword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using qword indices, the instruction conditionally loads up to 2 or 4 qword values from the VSIB addressing memory operand, and updates the lower half of the destination register. The upper 128 or 256 bits of the destination register are zero'ed with qword indices.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: For dword indices, the instruction will gather four dword values. For qword indices, the instruction will gather two values and zero 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 zero the upper 128 bits of the destination.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

```
DEST := SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB:[7:6];
DISP: optional 1, 4 byte displacement;
MASK := SRC3;

VPGATHERDD (VEX.128 version)
MASK[MAXVL-1:128] := 0;
FOR j := 0 to 3
    i := j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] := FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] := 0;
    FI;
ENDFOR
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[MAXVL-1:128] := 0;
```

VPGATHERQD (VEX.128 version)

```
MASK[MAXVL-1:64] := 0;
FOR j := 0 to 3
    i := j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] := FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] := 0;
    FI;
ENDFOR
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
DEST[MAXVL-1:64] := 0;
```

VPGATHERDD (VEX.256 version)

```
MASK[MAXVL-1:256] := 0;
FOR j := 0 to 7
    i := j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] := FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] := 0;
    FI;
ENDFOR
FOR j := 0 to 7
    i := j * 32;
    DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[i+31:i])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] := FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] := 0;
ENDFOR
DEST[MAXVL-1:256] := 0;
```

VPGATHERQD (VEX.256 version)

```
MASK[MAXVL-1:128] := 0;
FOR j := 0 to 7
    i := j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] := FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] := 0;
    FI;
ENDFOR
FOR j := 0 to 3
    k := j * 64;
    i := j * 32;
    DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] := FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] := 0;
ENDFOR
DEST[MAXVL-1:128] := 0;
```

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 Table 2-27, "Type 12 Class Exception Conditions."

VPGATHERDQ/VPGATHERQQ—Gather Packed Qword Values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.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.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.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.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)	N/A

Description

The instruction conditionally loads up to 2 or 4 qword values from memory addresses specified by the memory operand (the second operand) and using qword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using dword indices in the lower half of the mask register, the instruction conditionally loads up to 2 or 4 qword values from the VSIB addressing memory operand, and updates the destination register.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: The instruction will gather two qword values. For dword indices, only the lower two indices in the vector index register are used.

VEX.256 version: The instruction will gather four qword values. For dword indices, only the lower four indices in the vector index register are used.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

```
DEST := SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB:[7:6];
DISP: optional 1, 4 byte displacement;
MASK := SRC3;

VPGATHERDQ (VEX.128 version)
MASK[MAXVL-1:128] := 0;
FOR j := 0 to 1
    i := j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] := FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] := 0;
    FI;
ENDFOR
FOR j := 0 to 1
    k := j * 32;
    i := j * 64;
    DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX[k+31:k])*SCALE + DISP);
    IF MASK[63+i] THEN
        DEST[i +63:i] := FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63:i] := 0;
ENDFOR
DEST[MAXVL-1:128] := 0;
```

VPGATHERQQ (VEX.128 version)

```
MASK[MAXVL-1:128] := 0;
FOR j := 0 to 1
    i := j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] := FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] := 0;
    FI;
ENDFOR
FOR j := 0 to 1
    i := j * 64;
    DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP);
    IF MASK[63+i] THEN
        DEST[i +63:i] := FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63:i] := 0;
ENDFOR
DEST[MAXVL-1:128] := 0;
```

VPGATHERQQ (VEX.256 version)

```
MASK[MAXVL-1:256] := 0;
FOR j := 0 to 3
    i := j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] := FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] := 0;
    FI;
ENDFOR
FOR j := 0 to 3
    i := j * 64;
    DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP);
    IF MASK[63+i] THEN
        DEST[i +63:i] := FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63:i] := 0;
ENDFOR
DEST[MAXVL-1:256] := 0;
```

VPGATHERDQ (VEX.256 version)

```
MASK[MAXVL-1:256] := 0;
FOR j := 0 to 3
    i := j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] := FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] := 0;
    FI;
ENDFOR
FOR j := 0 to 3
    k := j * 32;
    i := j * 64;
    DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[k+31:k])*SCALE + DISP);
```

```
    IF MASK[63+i] THEN
        DEST[i +63:i] := FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63:i] := 0;
ENDFOR
DEST[MAXVL-1:256] := 0;
```

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 Table 2-27, "Type 12 Class Exception Conditions."

VPGATHERQD/VPGATHERQQ—Gather Packed Dword, Packed Qword with Signed Qword Indices

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 91 /vsib VPGATHERQD xmm1 {k1}, vm64x	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.256.66.0F38.W0 91 /vsib VPGATHERQD xmm1 {k1}, vm64y	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.512.66.0F38.W0 91 /vsib VPGATHERQD ymm1 {k1}, vm64z	A	V/V	AVX512F OR AVX10.1	Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.128.66.0F38.W1 91 /vsib VPGATHERQQ xmm1 {k1}, vm64x	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.
EVEX.256.66.0F38.W1 91 /vsib VPGATHERQQ ymm1 {k1}, vm64y	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.
EVEX.512.66.0F38.W1 91 /vsib VPGATHERQQ zmm1 {k1}, vm64z	A	V/V	AVX512F OR AVX10.1	Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	N/A	N/A

Description

A set of 8 doubleword/quadword memory locations pointed to by base address BASE_ADDR and index vector VINDEX with scale SCALE are gathered. The result is written into a vector register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.

- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- These instructions do not accept zeroing-masking since the 0 values in k1 are used to determine completion.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has the same disp8*N and alignment rules as for scalar instructions (Tuple 1).

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1 or 4 byte displacement

VPGATHERQD (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j]
        THEN DEST[i+31:i] := MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP]
            k1[j] := 0
        ELSE *DEST[i+31:i] := remains unchanged*            ; Only merging masking is allowed
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL/2] := 0
```

VPGATHERQQ (EVEX encoded version)

```
(KL, VL) = (2, 64), (4, 128), (8, 256)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j]
        THEN DEST[i+63:i] :=
            MEM[BASE_ADDR + (VINDEX[i+63:i]) * SCALE + DISP]
            k1[j] := 0
        ELSE *DEST[i+63:i] := remains unchanged*            ; Only merging masking is allowed
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPGATHERQD __m256i _mm512_i64gather_epi32(__m512i vdx, void * base, int scale);

VPGATHERQD __m256i _mm512_mask_i64gather_epi32lo(__m256i s, __mmask8 k, __m512i vdx, void * base, int scale);
VPGATHERQD __m128i _mm256_mask_i64gather_epi32lo(__m128i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERQD __m128i _mm_mask_i64gather_epi32(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERQQ __m512i _mm512_i64gather_epi64(__m512i vdx, void * base, int scale);
VPGATHERQQ __m512i _mm512_mask_i64gather_epi64(__m512i s, __mmask8 k, __m512i vdx, void * base, int scale);
VPGATHERQQ __m256i _mm256_mask_i64gather_epi64(__m256i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERQQ __m128i _mm_mask_i64gather_epi64(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, "Type E12 Class Exception Conditions."

VPLZCNTD/Q—Count the Number of Leading Zero Bits for Packed Dword, Packed Qword Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 44 /r VPLZCNTD xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512CD) OR AVX10.1	Count the number of leading zero bits in each dword element of xmm2/m128/m32bcst using writemask k1.
EVEX.256.66.0F38.W0 44 /r VPLZCNTD ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512VL AND AVX512CD) OR AVX10.1	Count the number of leading zero bits in each dword element of ymm2/m256/m32bcst using writemask k1.
EVEX.512.66.0F38.W0 44 /r VPLZCNTD zmm1 {k1}{z}, zmm2/m512/m32bcst	A	V/V	AVX512CD OR AVX10.1	Count the number of leading zero bits in each dword element of zmm2/m512/m32bcst using writemask k1.
EVEX.128.66.0F38.W1 44 /r VPLZCNTQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512CD) OR AVX10.1	Count the number of leading zero bits in each qword element of xmm2/m128/m64bcst using writemask k1.
EVEX.256.66.0F38.W1 44 /r VPLZCNTQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512CD) OR AVX10.1	Count the number of leading zero bits in each qword element of ymm2/m256/m64bcst using writemask k1.
EVEX.512.66.0F38.W1 44 /r VPLZCNTQ zmm1 {k1}{z}, zmm2/m512/m64bcst	A	V/V	AVX512CD OR AVX10.1	Count the number of leading zero bits in each qword element of zmm2/m512/m64bcst using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Counts the number of leading most significant zero bits in each dword or qword element of the source operand (the second operand) and stores the results in the destination register (the first operand) according to the writemask. If an element is zero, the result for that element is the operand size of the element.

EVEX.512 encoded version: The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPLZCNTD

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j*32
    IF MaskBit(j) OR *no writemask*
        THEN
            temp := 32
            DEST[i+31:i] := 0
            WHILE (temp > 0) AND (SRC[i+temp-1] = 0)
            DO
                temp := temp – 1
                DEST[i+31:i] := DEST[i+31:i] + 1
            OD
        ELSE
          IF *merging-masking*
            THEN *DEST[i+31:i] remains unchanged*
            ELSE DEST[i+31:i] := 0
          FI
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPLZCNTQ

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j*64
    IF MaskBit(j) OR *no writemask*
        THEN
            temp := 64
            DEST[i+63:i] := 0
            WHILE (temp > 0) AND (SRC[i+temp-1] = 0)
            DO
                temp := temp – 1
                DEST[i+63:i] := DEST[i+63:i] + 1
            OD
        ELSE
          IF *merging-masking*
            THEN *DEST[i+63:i] remains unchanged*
            ELSE DEST[i+63:i] := 0
          FI
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPLZCNTD __m512i _mm512_lzcnt_epi32(__m512i a);
VPLZCNTD __m512i _mm512_mask_lzcnt_epi32(__m512i s, __mmask16 m, __m512i a);
VPLZCNTD __m512i _mm512_maskz_lzcnt_epi32(__mmask16 m, __m512i a);
VPLZCNTQ __m512i _mm512_lzcnt_epi64(__m512i a);
VPLZCNTQ __m512i _mm512_mask_lzcnt_epi64(__m512i s, __mmask8 m, __m512i a);
VPLZCNTQ __m512i _mm512_maskz_lzcnt_epi64(__mmask8 m, __m512i a);
VPLZCNTD __m256i _mm256_lzcnt_epi32(__m256i a);
VPLZCNTD __m256i _mm256_mask_lzcnt_epi32(__m256i s, __mmask8 m, __m256i a);
VPLZCNTD __m256i _mm256_maskz_lzcnt_epi32(__mmask8 m, __m256i a);
VPLZCNTQ __m256i _mm256_lzcnt_epi64(__m256i a);
VPLZCNTQ __m256i _mm256_mask_lzcnt_epi64(__m256i s, __mmask8 m, __m256i a);
VPLZCNTQ __m256i _mm256_maskz_lzcnt_epi64(__mmask8 m, __m256i a);
VPLZCNTD __m128i _mm_lzcnt_epi32(__m128i a);
VPLZCNTD __m128i _mm_mask_lzcnt_epi32(__m128i s, __mmask8 m, __m128i a);
VPLZCNTD __m128i _mm_maskz_lzcnt_epi32(__mmask8 m, __m128i a);
VPLZCNTQ __m128i _mm_lzcnt_epi64(__m128i a);
VPLZCNTQ __m128i _mm_mask_lzcnt_epi64(__m128i s, __mmask8 m, __m128i a);
VPLZCNTQ __m128i _mm_maskz_lzcnt_epi64(__mmask8 m, __m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

VPMADD52HUQ—Packed Multiply of Unsigned 52-Bit Unsigned Integers and Add High 52-Bit Products to 64-Bit Accumulators

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 B5 /r VPMADD52HUQ xmm1, xmm2, xmm3/m128	A	V/V	AVX_IFMA	Multiply unsigned 52-bit integers in xmm2 and xmm3/m128 and add the high 52 bits of the 104-bit product to the qword unsigned integers in xmm1.
VEX.256.66.0F38.W1 B5 /r VPMADD52HUQ ymm1, ymm2, ymm3/m256	A	V/V	AVX_IFMA	Multiply unsigned 52-bit integers in ymm2 and ymm3/m256 and add the high 52 bits of the 104-bit product to the qword unsigned integers in ymm1.
EVEX.128.66.0F38.W1 B5 /r VPMADD52HUQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512_IFMA AND AVX512VL) OR AVX10.1	Multiply unsigned 52-bit integers in xmm2 and xmm3/m128 and add the high 52 bits of the 104-bit product to the qword unsigned integers in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 B5 /r VPMADD52HUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512_IFMA AND AVX512VL) OR AVX10.1	Multiply unsigned 52-bit integers in ymm2 and ymm3/m256 and add the high 52 bits of the 104-bit product to the qword unsigned integers in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 B5 /r VPMADD52HUQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512_IFMA OR AVX10.1	Multiply unsigned 52-bit integers in zmm2 and zmm3/m512 and add the high 52 bits of the 104-bit product to the qword unsigned integers in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies packed unsigned 52-bit integers in each qword element of the first source operand (the second operand) with the packed unsigned 52-bit integers in the corresponding elements of the second source operand (the third operand) to form packed 104-bit intermediate results. The high 52-bit, unsigned integer of each 104-bit product is added to the corresponding qword unsigned integer of the destination operand (the first operand) under the writemask k1.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 64-bit granularity.

Operation

VPMADDHUQ srcdest, src1, src2 (VEX version)

```
VL = (128,256)
KL = VL/64

FOR i in 0 .. KL-1:
    temp128 := zeroextend64(src1.qword[i][51:0]) *zeroextend64(src2.qword[i][51:0])
    srcdest.qword[i] := srcdest.qword[i] +zeroextend64(temp128[103:52])
srcdest[MAXVL:VL] := 0
```

VPMADD52HUQ (EVEX encoded)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64;
    IF k1[j] OR *no writemask* THEN
        IF src2 is Memory AND EVEX.b=1 THEN
            tsrc2[63:0] := ZeroExtend64(src2[51:0]);
        ELSE
            tsrc2[63:0] := ZeroExtend64(src2[i+51:i];
        FI;
        Temp128[127:0] := ZeroExtend64(src1[i+51:i]) * tsrc2[63:0];
        Temp2[63:0] := DEST[i+63:i] + ZeroExtend64(temp128[103:52]) ;
        DEST[i+63:i] := Temp2[63:0];
    ELSE
        IF *zeroing-masking* THEN
            DEST[i+63:i] := 0;
        ELSE *merge-masking*
            DEST[i+63:i] is unchanged;
        FI;
    FI;
ENDFOR
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMADD52HUQ __m128i _mm_madd52hi_avx_epu64 (__m128i __X, __m128i __Y, __m128i __Z);
VPMADD52HUQ __m128i _mm_maskz_madd52hi_epu64(__mmask8 k, __m128i a, __m128i b, __m128i c);
VPMADD52HUQ __m128i _mm_madd52hi_epu64 (__m128i __X, __m128i __Y, __m128i __Z);
VPMADD52HUQ __m128i _mm_madd52hi_epu64(__m128i a, __m128i b, __m128i c);
VPMADD52HUQ __m128i _mm_mask_madd52hi_epu64(__m128i s, __mmask8 k, __m128i a, __m128i b, __m128i c);
VPMADD52HUQ __m256i _mm256_madd52hi_avx_epu64 (__m256i __X, __m256i __Y, __m256i __Z);
VPMADD52HUQ __m256i _mm256_madd52hi_epu64(__m256i a, __m256i b, __m256i c);
VPMADD52HUQ __m256i _mm256_madd52hi_epu64 (__m256i __X, __m256i __Y, __m256i __Z);
VPMADD52HUQ __m256i _mm256_mask_madd52hi_epu64(__m256i s, __mmask8 k, __m256i a, __m256i b, __m256i c);
VPMADD52HUQ __m256i _mm256_maskz_madd52hi_epu64(__mmask8 k, __m256i a, __m256i b, __m256i c);
VPMADD52HUQ __m512i _mm512_madd52hi_epu64(__m512i a, __m512i b, __m512i c);
VPMADD52HUQ __m512i _mm512_mask_madd52hi_epu64(__m512i s, __mmask8 k, __m512i a, __m512i b, __m512i c);
VPMADD52HUQ __m512i _mm512_maskz_madd52hi_epu64(__mmask8 k, __m512i a, __m512i b, __m512i c);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-51, “Type E4 Class Exception Conditions.”

VPMADD52LUQ—Packed Multiply of Unsigned 52-Bit Integers and Add the Low 52-Bit Products to Qword Accumulators

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 B4 /r VPMADD52LUQ xmm1, xmm2, xmm3/m128	A	V/V	AVX_IFMA	Multiply unsigned 52-bit integers in xmm2 and xmm3/m128 and add the low 52 bits of the 104-bit product to the qword unsigned integers in xmm1.
VEX.256.66.0F38.W1 B4 /r VPMADD52LUQ ymm1, ymm2, ymm3/m256	A	V/V	AVX_IFMA	Multiply unsigned 52-bit integers in ymm2 and ymm3/m256 and add the low 52 bits of the 104-bit product to the qword unsigned integers in ymm1.
EVEX.128.66.0F38.W1 B4 /r VPMADD52LUQ xmm1 {k1}{z}, xmm2,xmm3/m128/m64bcst	B	V/V	(AVX512_IFMA AND AVX512VL) OR AVX10.1	Multiply unsigned 52-bit integers in xmm2 and xmm3/m128 and add the low 52 bits of the 104-bit product to the qword unsigned integers in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 B4 /r VPMADD52LUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512_IFMA AND AVX512VL) OR AVX10.1	Multiply unsigned 52-bit integers in ymm2 and ymm3/m256 and add the low 52 bits of the 104-bit product to the qword unsigned integers in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 B4 /r VPMADD52LUQ zmm1 {k1}{z}, zmm2,zmm3/m512/m64bcst	B	V/V	AVX512_IFMA OR AVX10.1	Multiply unsigned 52-bit integers in zmm2 and zmm3/m512 and add the low 52 bits of the 104-bit product to the qword unsigned integers in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m(r)	N/A

Description

Multiplies packed unsigned 52-bit integers in each qword element of the first source operand (the second operand) with the packed unsigned 52-bit integers in the corresponding elements of the second source operand (the third operand) to form packed 104-bit intermediate results. The low 52-bit, unsigned integer of each 104-bit product is added to the corresponding qword unsigned integer of the destination operand (the first operand) under the writemask k1.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 64-bit granularity.

Operation

VPMADDLUQ srcdest, src1, src2 (VEX version)

```
VL = (128,256)
KL = VL/64

FOR i in 0 .. KL-1:
    temp128 := zeroextend64(src1.qword[i][51:0]) *zeroextend64(src2.qword[i][51:0])
    srcdest.qword[i] := srcdest.qword[i] +zeroextend64(temp128[51:0])
srcdest[MAXVL:VL] := 0
```

VPMADD52LUQ (EVEX encoded)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64;
    IF k1[j] OR *no writemask* THEN
        IF src2 is Memory AND EVEX.b=1 THEN
            tsrc2[63:0] := ZeroExtend64(src2[51:0]);
        ELSE
            tsrc2[63:0] := ZeroExtend64(src2[i+51:i];
        FI;
        Temp128[127:0] := ZeroExtend64(src1[i+51:i]) * tsrc2[63:0];
        Temp2[63:0] := DEST[i+63:i] + ZeroExtend64(temp128[51:0]) ;
        DEST[i+63:i] := Temp2[63:0];
    ELSE
        IF *zeroing-masking* THEN
            DEST[i+63:i] := 0;
        ELSE *merge-masking*
            DEST[i+63:i] is unchanged;
        FI;
    FI;
ENDFOR
DEST[MAX_VL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMADD52LUQ __m128i _mm_madd52lo_avx_epu64 (__m128i __X, __m128i __Y, __m128i __Z);
VPMADD52LUQ __m128i _mm_madd52lo_epu64(__m128i a, __m128i b, __m128i c);
VPMADD52LUQ __m128i _mm_madd52lo_epu64 (__m128i __X, __m128i __Y, __m128i __Z);
VPMADD52LUQ __m128i _mm_mask_madd52lo_epu64(__m128i s, __mmask8 k, __m128i a, __m128i b, __m128i c);
VPMADD52LUQ __m128i _mm_maskz_madd52lo_epu64(__mmask8 k, __m128i a, __m128i b, __m128i c);
VPMADD52LUQ __m256i _mm256_madd52lo_avx_epu64 (__m256i __X, __m256i __Y, __m256i __Z);
VPMADD52LUQ __m256i _mm256_madd52lo_epu64(__m256i a, __m256i b, __m256i c);
VPMADD52LUQ __m256i _mm256_madd52lo_epu64 (__m256i __X, __m256i __Y, __m256i __Z);
VPMADD52LUQ __m256i _mm256_mask_madd52lo_epu64(__m256i s, __mmask8 k, __m256i a, __m256i b, __m256i c);
VPMADD52LUQ __m256i _mm256_maskz_madd52lo_epu64(__mmask8 k, __m256i a, __m256i b, __m256i c);
VPMADD52LUQ __m512i _mm512_madd52lo_epu64(__m512i a, __m512i b, __m512i c);
VPMADD52LUQ __m512i _mm512_mask_madd52lo_epu64(__m512i s, __mmask8 k, __m512i a, __m512i b, __m512i c);
VPMADD52LUQ __m512i _mm512_maskz_madd52lo_epu64(__mmask8 k, __m512i a, __m512i b, __m512i c);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-51, “Type E4 Class Exception Conditions.”

VPMASKMOV—Conditional SIMD Integer Packed Loads and Stores

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 8C /r VPMASKMOVD xmm1, xmm2, m128	RVM	V/V	AVX2	Conditionally load dword values from m128 using mask in xmm2 and store in xmm1.
VEX.256.66.0F38.W0 8C /r VPMASKMOVD ymm1, ymm2, m256	RVM	V/V	AVX2	Conditionally load dword values from m256 using mask in ymm2 and store in ymm1.
VEX.128.66.0F38.W1 8C /r VPMASKMOVQ xmm1, xmm2, m128	RVM	V/V	AVX2	Conditionally load qword values from m128 using mask in xmm2 and store in xmm1.
VEX.256.66.0F38.W1 8C /r VPMASKMOVQ ymm1, ymm2, m256	RVM	V/V	AVX2	Conditionally load qword values from m256 using mask in ymm2 and store in ymm1.
VEX.128.66.0F38.W0 8E /r VPMASKMOVD m128, xmm1, xmm2	MVR	V/V	AVX2	Conditionally store dword values from xmm2 using mask in xmm1.
VEX.256.66.0F38.W0 8E /r VPMASKMOVD m256, ymm1, ymm2	MVR	V/V	AVX2	Conditionally store dword values from ymm2 using mask in ymm1.
VEX.128.66.0F38.W1 8E /r VPMASKMOVQ m128, xmm1, xmm2	MVR	V/V	AVX2	Conditionally store qword values from xmm2 using mask in xmm1.
VEX.256.66.0F38.W1 8E /r VPMASKMOVQ m256, ymm1, ymm2	MVR	V/V	AVX2	Conditionally store qword values from ymm2 using mask in ymm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
MVR	ModRM:r/m (w)	VEX.vvvv (r)	ModRM:reg (r)	N/A

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[MAXVL-1:128] := 0
```

VPMASKMOVQ - 256-bit load

```
DEST[63:0] := IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] := IF (SRC1[127]) Load_64(mem + 8) ELSE 0
DEST[195:128] := IF (SRC1[191]) Load_64(mem + 16) ELSE 0
DEST[255:196] := IF (SRC1[255]) Load_64(mem + 24) ELSE 0
```

VPMASKMOVQ - 128-bit load

```
DEST[63:0] := IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] := IF (SRC1[127]) Load_64(mem + 8) ELSE 0
DEST[MAXVL-1:128] := 0
```

VPMASKMOVD - 256-bit store

```
IF (SRC1[31]) DEST[31:0] := SRC2[31:0]
IF (SRC1[63]) DEST[63:32] := SRC2[63:32]
IF (SRC1[95]) DEST[95:64] := SRC2[95:64]
IF (SRC1[127]) DEST[127:96] := SRC2[127:96]
IF (SRC1[159]) DEST[159:128] :=SRC2[159:128]
IF (SRC1[191]) DEST[191:160] := SRC2[191:160]
IF (SRC1[223]) DEST[223:192] := SRC2[223:192]
IF (SRC1[255]) DEST[255:224] := SRC2[255:224]
```

VPMASKMOVD - 128-bit store
IF (SRC1[31]) DEST[31:0] := SRC2[31:0]
IF (SRC1[63]) DEST[63:32] := SRC2[63:32]
IF (SRC1[95]) DEST[95:64] := SRC2[95:64]
IF (SRC1[127]) DEST[127:96] := SRC2[127:96]

VPMASKMOVQ - 256-bit store
IF (SRC1[63]) DEST[63:0] := SRC2[63:0]
IF (SRC1[127]) DEST[127:64] :=SRC2[127:64]
IF (SRC1[191]) DEST[191:128] := SRC2[191:128]
IF (SRC1[255]) DEST[255:192] := SRC2[255:192]

VPMASKMOVQ - 128-bit store
IF (SRC1[63]) DEST[63:0] := SRC2[63:0]
IF (SRC1[127]) DEST[127:64] :=SRC2[127:64]

Intel C/C++ Compiler Intrinsic Equivalent

VPMASKMOVD: __m256i _mm256_maskload_epi32(int const *a, __m256i mask)

VPMASKMOVD: void _mm256_maskstore_epi32(int *a, __m256i mask, __m256i b)

VPMASKMOVQ: __m256i _mm256_maskload_epi64(__int64 const *a, __m256i mask);

VPMASKMOVQ: void _mm256_maskstore_epi64(__int64 *a, __m256i mask, __m256d b);

VPMASKMOVD: __m128i _mm_maskload_epi32(int const *a, __m128i mask)

VPMASKMOVD: void _mm_maskstore_epi32(int *a, __m128i mask, __m128 b)

VPMASKMOVQ: __m128i _mm_maskload_epi64(__int cont *a, __m128i mask);

VPMASKMOVQ: void _mm_maskstore_epi64(__int64 *a, __m128i mask, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-23, "Type 6 Class Exception Conditions" (No AC# reported for any mask bit combinations).

VPMOVB2M/VPMOVW2M/VPMOVD2M/VPMOVQ2M—Convert a Vector Register to a Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 29 /r VPMOVB2M k1, xmm1	RM	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in XMM1.
EVEX.256.F3.0F38.W0 29 /r VPMOVB2M k1, ymm1	RM	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in YMM1.
EVEX.512.F3.0F38.W0 29 /r VPMOVB2M k1, zmm1	RM	V/V	AVX512BW OR AVX10.1	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in ZMM1.
EVEX.128.F3.0F38.W1 29 /r VPMOVW2M k1, xmm1	RM	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in XMM1.
EVEX.256.F3.0F38.W1 29 /r VPMOVW2M k1, ymm1	RM	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in YMM1.
EVEX.512.F3.0F38.W1 29 /r VPMOVW2M k1, zmm1	RM	V/V	AVX512BW OR AVX10.1	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in ZMM1.
EVEX.128.F3.0F38.W0 39 /r VPMOVD2M k1, xmm1	RM	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in XMM1.
EVEX.256.F3.0F38.W0 39 /r VPMOVD2M k1, ymm1	RM	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in YMM1.
EVEX.512.F3.0F38.W0 39 /r VPMOVD2M k1, zmm1	RM	V/V	AVX512DQ OR AVX10.1	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in ZMM1.
EVEX.128.F3.0F38.W1 39 /r VPMOVQ2M k1, xmm1	RM	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in XMM1.
EVEX.256.F3.0F38.W1 39 /r VPMOVQ2M k1, ymm1	RM	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in YMM1.
EVEX.512.F3.0F38.W1 39 /r VPMOVQ2M k1, zmm1	RM	V/V	AVX512DQ OR AVX10.1	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in ZMM1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts a vector register to a mask register. Each element in the destination register is set to 1 or 0 depending on the value of most significant bit of the corresponding element in the source register.

The source operand is a ZMM/YMM/XMM register. The destination operand is a mask register.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVB2M (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF SRC[i+7]
        THEN    DEST[j] := 1
        ELSE    DEST[j] := 0
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPMOVW2M (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF SRC[i+15]
        THEN    DEST[j] := 1
        ELSE    DEST[j] := 0
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPMOVD2M (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF SRC[i+31]
        THEN    DEST[j] := 1
        ELSE    DEST[j] := 0
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPMOVQ2M (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF SRC[i+63]
        THEN    DEST[j] := 1
        ELSE    DEST[j] := 0
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMPOVB2M __mmask64 _mm512_movepi8_mask(__m512i);
VPMPOVD2M __mmask16 _mm512_movepi32_mask(__m512i);
VPMPOVQ2M __mmask8 _mm512_movepi64_mask(__m512i);
VPMPOVW2M __mmask32 _mm512_movepi16_mask(__m512i);
VPMPOVB2M __mmask32 _mm256_movepi8_mask(__m256i);
VPMPOVD2M __mmask8 _mm256_movepi32_mask(__m256i);
VPMPOVQ2M __mmask8 _mm256_movepi64_mask(__m256i);
VPMPOVW2M __mmask16 _mm256_movepi16_mask(__m256i);
VPMPOVB2M __mmask16 _mm_movepi8_mask(__m128i);
VPMPOVD2M __mmask8 _mm_movepi32_mask(__m128i);
VPMPOVQ2M __mmask8 _mm_movepi64_mask(__m128i);
VPMPOVW2M __mmask8 _mm_movepi16_mask(__m128i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-57, "Type E7NM Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VPMOVDB/VPMOVSDB/VPMOVUSDB—Down Convert DWord to Byte

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 31 /r VPMOVDB xmm1/m32 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed double-word integers from xmm2 into 4 packed byte integers in xmm1/m32 with truncation under writemask k1.
EVEX.128.F3.0F38.W0 21 /r VPMOVSDB xmm1/m32 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed signed double-word integers from xmm2 into 4 packed signed byte integers in xmm1/m32 using signed saturation under writemask k1.
EVEX.128.F3.0F38.W0 11 /r VPMOVUSDB xmm1/m32 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed unsigned double-word integers from xmm2 into 4 packed unsigned byte integers in xmm1/m32 using unsigned saturation under writemask k1.
EVEX.256.F3.0F38.W0 31 /r VPMOVDB xmm1/m64 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 8 packed double-word integers from ymm2 into 8 packed byte integers in xmm1/m64 with truncation under writemask k1.
EVEX.256.F3.0F38.W0 21 /r VPMOVSDB xmm1/m64 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 8 packed signed double-word integers from ymm2 into 8 packed signed byte integers in xmm1/m64 using signed saturation under writemask k1.
EVEX.256.F3.0F38.W0 11 /r VPMOVUSDB xmm1/m64 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 8 packed unsigned double-word integers from ymm2 into 8 packed unsigned byte integers in xmm1/m64 using unsigned saturation under writemask k1.
EVEX.512.F3.0F38.W0 31 /r VPMOVDB xmm1/m128 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 16 packed double-word integers from zmm2 into 16 packed byte integers in xmm1/m128 with truncation under writemask k1.
EVEX.512.F3.0F38.W0 21 /r VPMOVSDB xmm1/m128 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 16 packed signed double-word integers from zmm2 into 16 packed signed byte integers in xmm1/m128 using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 11 /r VPMOVUSDB xmm1/m128 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 16 packed unsigned double-word integers from zmm2 into 16 packed unsigned byte integers in xmm1/m128 using unsigned saturation under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Quarter Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

VPMOVDB down converts 32-bit integer elements in the source operand (the second operand) into packed bytes using truncation. VPMOVSDB converts signed 32-bit integers into packed signed bytes using signed saturation. VPMOVUSDB convert unsigned double-word values into unsigned byte values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a XMM register or a 128/64/32-bit memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:128/64/32) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVDB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 8
    m := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TruncateDoubleWordToByte (SRC[m+31:m])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/4] := 0;
```

VPMOVDB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 8
    m := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TruncateDoubleWordToByte (SRC[m+31:m])
        ELSE *DEST[i+7:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR
```

VPMOVSDB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 8
    m := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SaturateSignedDoubleWordToByte (SRC[m+31:m])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
```

```
        FI;
    ENDFOR
    DEST[MAXVL-1:VL/4] := 0;
```

VPMOVSDB instruction (EVEX encoded versions) when dest is memory

```
    (KL, VL) = (4, 128), (8, 256), (16, 512)
    FOR j := 0 TO KL-1
        i := j * 8
        m := j * 32
        IF k1[j] OR *no writemask*
            THEN DEST[i+7:i] := SaturateSignedDoubleWordToByte (SRC[m+31:m])
            ELSE *DEST[i+7:i] remains unchanged*             ; merging-masking
        FI;
    ENDFOR
```

VPMOVUSDB instruction (EVEX encoded versions) when dest is a register

```
    (KL, VL) = (4, 128), (8, 256), (16, 512)
    FOR j := 0 TO KL-1
        i := j * 8
        m := j * 32
        IF k1[j] OR *no writemask*
            THEN DEST[i+7:i] := SaturateUnsignedDoubleWordToByte (SRC[m+31:m])
            ELSE
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+7:i] remains unchanged*
                    ELSE *zeroing-masking*              ; zeroing-masking
                        DEST[i+7:i] := 0
                FI
        FI;
    ENDFOR
    DEST[MAXVL-1:VL/4] := 0;
```

VPMOVUSDB instruction (EVEX encoded versions) when dest is memory

```
    (KL, VL) = (4, 128), (8, 256), (16, 512)
    FOR j := 0 TO KL-1
        i := j * 8
        m := j * 32
        IF k1[j] OR *no writemask*
            THEN DEST[i+7:i] := SaturateUnsignedDoubleWordToByte (SRC[m+31:m])
            ELSE *DEST[i+7:i] remains unchanged*             ; merging-masking
        FI;
    ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMOVDB __m128i _mm512_cvtepi32_epi8(__m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVDB void _mm512_mask_cvtepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVSDB __m128i _mm512_cvtsepi32_epi8(__m512i a);
VPMOVSDB __m128i _mm512_mask_cvtsepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVSDB __m128i _mm512_maskz_cvtsepi32_epi8(__mmask16 k, __m512i a);
VPMOVSDB void _mm512_mask_cvtsepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVUSDB __m128i _mm512_cvtusepi32_epi8(__m512i a);
VPMOVUSDB __m128i _mm512_mask_cvtusepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVUSDB __m128i _mm512_maskz_cvtusepi32_epi8(__mmask16 k, __m512i a);
VPMOVUSDB void _mm512_mask_cvtusepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVUSDB __m128i _mm256_cvtusepi32_epi8(__m256i a);
VPMOVUSDB __m128i _mm256_mask_cvtusepi32_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVUSDB __m128i _mm256_maskz_cvtusepi32_epi8(__mmask8 k, __m256i b);
VPMOVUSDB void _mm256_mask_cvtusepi32_storeu_epi8(void * , __mmask8 k, __m256i b);
VPMOVUSDB __m128i _mm_cvtusepi32_epi8(__m128i a);
VPMOVUSDB __m128i _mm_mask_cvtusepi32_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVUSDB __m128i _mm_maskz_cvtusepi32_epi8(__mmask8 k, __m128i b);
VPMOVUSDB void _mm_mask_cvtusepi32_storeu_epi8(void * , __mmask8 k, __m128i b);
VPMOVSDB __m128i _mm256_cvtsepi32_epi8(__m256i a);
VPMOVSDB __m128i _mm256_mask_cvtsepi32_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVSDB __m128i _mm256_maskz_cvtsepi32_epi8(__mmask8 k, __m256i b);
VPMOVSDB void _mm256_mask_cvtsepi32_storeu_epi8(void * , __mmask8 k, __m256i b);
VPMOVSDB __m128i _mm_cvtsepi32_epi8(__m128i a);
VPMOVSDB __m128i _mm_mask_cvtsepi32_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVSDB __m128i _mm_maskz_cvtsepi32_epi8(__mmask8 k, __m128i b);
VPMOVSDB void _mm_mask_cvtsepi32_storeu_epi8(void * , __mmask8 k, __m128i b);
VPMOVDB __m128i _mm256_cvtepi32_epi8(__m256i a);
VPMOVDB __m128i _mm256_mask_cvtepi32_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVDB __m128i _mm256_maskz_cvtepi32_epi8(__mmask8 k, __m256i b);
VPMOVDB void _mm256_mask_cvtepi32_storeu_epi8(void * , __mmask8 k, __m256i b);
VPMOVDB __m128i _mm_cvtepi32_epi8(__m128i a);
VPMOVDB __m128i _mm_mask_cvtepi32_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVDB __m128i _mm_maskz_cvtepi32_epi8(__mmask8 k, __m128i b);
VPMOVDB void _mm_mask_cvtepi32_storeu_epi8(void * , __mmask8 k, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-55, "Type E6 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VPMOVDW/VPMOVSDW/VPMOVUSDW—Down Convert DWord to Word

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 33 /r VPMOVDW xmm1/m64 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed double-word integers from xmm2 into 4 packed word integers in xmm1/m64 with truncation under writemask k1.
EVEX.128.F3.0F38.W0 23 /r VPMOVSDW xmm1/m64 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed signed double-word integers from xmm2 into 4 packed signed word integers in ymm1/m64 using signed saturation under writemask k1.
EVEX.128.F3.0F38.W0 13 /r VPMOVUSDW xmm1/m64 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed unsigned double-word integers from xmm2 into 4 packed unsigned word integers in xmm1/m64 using unsigned saturation under writemask k1.
EVEX.256.F3.0F38.W0 33 /r VPMOVDW xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 8 packed double-word integers from ymm2 into 8 packed word integers in xmm1/m128 with truncation under writemask k1.
EVEX.256.F3.0F38.W0 23 /r VPMOVSDW xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 8 packed signed double-word integers from ymm2 into 8 packed signed word integers in xmm1/m128 using signed saturation under writemask k1.
EVEX.256.F3.0F38.W0 13 /r VPMOVUSDW xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 8 packed unsigned double-word integers from ymm2 into 8 packed unsigned word integers in xmm1/m128 using unsigned saturation under writemask k1.
EVEX.512.F3.0F38.W0 33 /r VPMOVDW ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 16 packed double-word integers from zmm2 into 16 packed word integers in ymm1/m256 with truncation under writemask k1.
EVEX.512.F3.0F38.W0 23 /r VPMOVSDW ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 16 packed signed double-word integers from zmm2 into 16 packed signed word integers in ymm1/m256 using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 13 /r VPMOVUSDW ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 16 packed unsigned double-word integers from zmm2 into 16 packed unsigned word integers in ymm1/m256 using unsigned saturation under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

VPMOVDW down converts 32-bit integer elements in the source operand (the second operand) into packed words using truncation. VPMOVSDW converts signed 32-bit integers into packed signed words using signed saturation. VPMOVUSDW convert unsigned double-word values into unsigned word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted word elements are written to the destination operand (the first operand) from the least-significant word. Word elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVDW instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 16
    m := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TruncateDoubleWordToWord (SRC[m+31:m])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0;
```

VPMOVDW instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 16
    m := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TruncateDoubleWordToWord (SRC[m+31:m])
        ELSE
            *DEST[i+15:i] remains unchanged*    ; merging-masking
    FI;
ENDFOR
```

VPMOVSDW instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 16
    m := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SaturateSignedDoubleWordToWord (SRC[m+31:m])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] := 0
```

```
                FI
        FI;
    ENDFOR
    DEST[MAXVL-1:VL/2] := 0;
```

VPMOVSDW instruction (EVEX encoded versions) when dest is memory

```
    (KL, VL) = (4, 128), (8, 256), (16, 512)
    FOR j := 0 TO KL-1
        i := j * 16
        m := j * 32
        IF k1[j] OR *no writemask*
            THEN DEST[i+15:i] := SaturateSignedDoubleWordToWord (SRC[m+31:m])
            ELSE
                *DEST[i+15:i] remains unchanged*     ; merging-masking
        FI;
    ENDFOR
```

VPMOVUSDW instruction (EVEX encoded versions) when dest is a register

```
    (KL, VL) = (4, 128), (8, 256), (16, 512)
    FOR j := 0 TO KL-1
        i := j * 16
        m := j * 32
        IF k1[j] OR *no writemask*
            THEN DEST[i+15:i] := SaturateUnsignedDoubleWordToWord (SRC[m+31:m])
            ELSE
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+15:i] remains unchanged*
                    ELSE *zeroing-masking*              ; zeroing-masking
                        DEST[i+15:i] := 0
                FI
        FI;
    ENDFOR
    DEST[MAXVL-1:VL/2] := 0;
```

VPMOVUSDW instruction (EVEX encoded versions) when dest is memory

```
    (KL, VL) = (4, 128), (8, 256), (16, 512)
    FOR j := 0 TO KL-1
        i := j * 16
        m := j * 32
        IF k1[j] OR *no writemask*
            THEN DEST[i+15:i] := SaturateUnsignedDoubleWordToWord (SRC[m+31:m])
            ELSE
                *DEST[i+15:i] remains unchanged*     ; merging-masking
        FI;
    ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMOVDW __m256i _mm512_cvtepi32_epi16(__m512i a);
VPMOVDW __m256i _mm512_mask_cvtepi32_epi16(__m256i s, __mmask16 k, __m512i a);
VPMOVDW __m256i _mm512_maskz_cvtepi32_epi16(__mmask16 k, __m512i a);
VPMOVDW void _mm512_mask_cvtepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVSDW __m256i _mm512_cvtsepi32_epi16(__m512i a);
VPMOVSDW __m256i _mm512_mask_cvtsepi32_epi16(__m256i s, __mmask16 k, __m512i a);
VPMOVSDW __m256i _mm512_maskz_cvtsepi32_epi16(__mmask16 k, __m512i a);
VPMOVSDW void _mm512_mask_cvtsepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVUSDW __m256i _mm512_cvtusepi32_epi16 __m512i a);
VPMOVUSDW __m256i _mm512_mask_cvtusepi32_epi16(__m256i s, __mmask16 k, __m512i a);
VPMOVUSDW __m256i _mm512_maskz_cvtusepi32_epi16(__mmask16 k, __m512i a);
VPMOVUSDW void _mm512_mask_cvtusepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVUSDW __m128i _mm256_cvtusepi32_epi16(__m256i a);
VPMOVUSDW __m128i _mm256_mask_cvtusepi32_epi16(__m128i a, __mmask8 k, __m256i b);
VPMOVUSDW __m128i _mm256_maskz_cvtusepi32_epi16(__mmask8 k, __m256i b);
VPMOVUSDW void _mm256_mask_cvtusepi32_storeu_epi16(void * , __mmask8 k, __m256i b);
VPMOVUSDW __m128i _mm_cvtusepi32_epi16(__m128i a);
VPMOVUSDW __m128i _mm_mask_cvtusepi32_epi16(__m128i a, __mmask8 k, __m128i b);
VPMOVUSDW __m128i _mm_maskz_cvtusepi32_epi16(__mmask8 k, __m128i b);
VPMOVUSDW void _mm_mask_cvtusepi32_storeu_epi16(void * , __mmask8 k, __m128i b);
VPMOVSDW __m128i _mm256_cvtsepi32_epi16(__m256i a);
VPMOVSDW __m128i _mm256_mask_cvtsepi32_epi16(__m128i a, __mmask8 k, __m256i b);
VPMOVSDW __m128i _mm256_maskz_cvtsepi32_epi16(__mmask8 k, __m256i b);
VPMOVSDW void _mm256_mask_cvtsepi32_storeu_epi16(void * , __mmask8 k, __m256i b);
VPMOVSDW __m128i _mm_cvtsepi32_epi16(__m128i a);
VPMOVSDW __m128i _mm_mask_cvtsepi32_epi16(__m128i a, __mmask8 k, __m128i b);
VPMOVSDW __m128i _mm_maskz_cvtsepi32_epi16(__mmask8 k, __m128i b);
VPMOVSDW void _mm_mask_cvtsepi32_storeu_epi16(void * , __mmask8 k, __m128i b);
VPMOVDW __m128i _mm256_cvtepi32_epi16(__m256i a);
VPMOVDW __m128i _mm256_mask_cvtepi32_epi16(__m128i a, __mmask8 k, __m256i b);
VPMOVDW __m128i _mm256_maskz_cvtepi32_epi16(__mmask8 k, __m256i b);
VPMOVDW void _mm256_mask_cvtepi32_storeu_epi16(void * , __mmask8 k, __m256i b);
VPMOVDW __m128i _mm_cvtepi32_epi16(__m128i a);
VPMOVDW __m128i _mm_mask_cvtepi32_epi16(__m128i a, __mmask8 k, __m128i b);
VPMOVDW __m128i _mm_maskz_cvtepi32_epi16(__mmask8 k, __m128i b);
VPMOVDW void _mm_mask_cvtepi32_storeu_epi16(void * , __mmask8 k, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-55, "Type E6 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VPMOVM2B/VPMOVM2W/VPMOVM2D/VPMOVM2Q—Convert a Mask Register to a Vector Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 28 /r VPMOVM2B xmm1, k1	RM	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Sets each byte in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.256.F3.0F38.W0 28 /r VPMOVM2B ymm1, k1	RM	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Sets each byte in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.512.F3.0F38.W0 28 /r VPMOVM2B zmm1, k1	RM	V/V	AVX512BW OR AVX10.1	Sets each byte in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.128.F3.0F38.W1 28 /r VPMOVM2W xmm1, k1	RM	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Sets each word in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.256.F3.0F38.W1 28 /r VPMOVM2W ymm1, k1	RM	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Sets each word in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.512.F3.0F38.W1 28 /r VPMOVM2W zmm1, k1	RM	V/V	AVX512BW OR AVX10.1	Sets each word in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.128.F3.0F38.W0 38 /r VPMOVM2D xmm1, k1	RM	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Sets each doubleword in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.256.F3.0F38.W0 38 /r VPMOVM2D ymm1, k1	RM	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Sets each doubleword in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.512.F3.0F38.W0 38 /r VPMOVM2D zmm1, k1	RM	V/V	AVX512DQ OR AVX10.1	Sets each doubleword in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.128.F3.0F38.W1 38 /r VPMOVM2Q xmm1, k1	RM	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Sets each quadword in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.256.F3.0F38.W1 38 /r VPMOVM2Q ymm1, k1	RM	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Sets each quadword in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.512.F3.0F38.W1 38 /r VPMOVM2Q zmm1, k1	RM	V/V	AVX512DQ OR AVX10.1	Sets each quadword in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts a mask register to a vector register. Each element in the destination register is set to all 1's or all 0's depending on the value of the corresponding bit in the source mask register.

The source operand is a mask register. The destination operand is a ZMM/YMM/XMM register.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVM2B (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF SRC[j]
        THEN    DEST[i+7:i] := -1
        ELSE    DEST[i+7:i] := 0
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVM2W (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF SRC[j]
        THEN    DEST[i+15:i] := -1
        ELSE    DEST[i+15:i] := 0
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVM2D (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF SRC[j]
        THEN    DEST[i+31:i] := -1
        ELSE    DEST[i+31:i] := 0
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVM2Q (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF SRC[j]
        THEN    DEST[i+63:i] := -1
        ELSE    DEST[i+63:i] := 0
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMOVM2B __m512i _mm512_movm_epi8(__mmask64);
VPMOVM2D __m512i _mm512_movm_epi32(__mmask8);
VPMOVM2Q __m512i _mm512_movm_epi64(__mmask16);
VPMOVM2W __m512i _mm512_movm_epi16(__mmask32);
VPMOVM2B __m256i _mm256_movm_epi8(__mmask32);
VPMOVM2D __m256i _mm256_movm_epi32(__mmask8);
VPMOVM2Q __m256i _mm256_movm_epi64(__mmask8);
VPMOVM2W __m256i _mm256_movm_epi16(__mmask16);
VPMOVM2B __m128i _mm_movm_epi8(__mmask16);
VPMOVM2D __m128i _mm_movm_epi32(__mmask8);
VPMOVM2Q __m128i _mm_movm_epi64(__mmask8);
VPMOVM2W __m128i _mm_movm_epi16(__mmask8);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-57, "Type E7NM Class Exception Conditions."
Additionally:

#UD	If EVEX.vvvv != 1111B.

VPMOVQB/VPMOVSQB/VPMOVUSQB—Down Convert QWord to Byte

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 32 /r VPMOVQB xmm1/m16 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 2 packed quad-word integers from xmm2 into 2 packed byte integers in xmm1/m16 with truncation under writemask k1.
EVEX.128.F3.0F38.W0 22 /r VPMOVSQB xmm1/m16 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 2 packed signed quad-word integers from xmm2 into 2 packed signed byte integers in xmm1/m16 using signed saturation under writemask k1.
EVEX.128.F3.0F38.W0 12 /r VPMOVUSQB xmm1/m16 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 2 packed unsigned quad-word integers from xmm2 into 2 packed unsigned byte integers in xmm1/m16 using unsigned saturation under writemask k1.
EVEX.256.F3.0F38.W0 32 /r VPMOVQB xmm1/m32 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed quad-word integers from ymm2 into 4 packed byte integers in xmm1/m32 with truncation under writemask k1.
EVEX.256.F3.0F38.W0 22 /r VPMOVSQB xmm1/m32 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed signed quad-word integers from ymm2 into 4 packed signed byte integers in xmm1/m32 using signed saturation under writemask k1.
EVEX.256.F3.0F38.W0 12 /r VPMOVUSQB xmm1/m32 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed unsigned quad-word integers from ymm2 into 4 packed unsigned byte integers in xmm1/m32 using unsigned saturation under writemask k1.
EVEX.512.F3.0F38.W0 32 /r VPMOVQB xmm1/m64 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 8 packed quad-word integers from zmm2 into 8 packed byte integers in xmm1/m64 with truncation under writemask k1.
EVEX.512.F3.0F38.W0 22 /r VPMOVSQB xmm1/m64 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 8 packed signed quad-word integers from zmm2 into 8 packed signed byte integers in xmm1/m64 using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 12 /r VPMOVUSQB xmm1/m64 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 8 packed unsigned quad-word integers from zmm2 into 8 packed unsigned byte integers in xmm1/m64 using unsigned saturation under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Eighth Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

VPMOVQB down converts 64-bit integer elements in the source operand (the second operand) into packed byte elements using truncation. VPMOVSQB converts signed 64-bit integers into packed signed bytes using signed saturation. VPMOVUSQB convert unsigned quad-word values into unsigned byte values using unsigned saturation. The source operand is a vector register. The destination operand is an XMM register or a memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:64) of the destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVQB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 8
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TruncateQuadWordToByte (SRC[m+63:m])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/8] := 0;
```

VPMOVQB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 8
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TruncateQuadWordToByte (SRC[m+63:m])
        ELSE
            *DEST[i+7:i] remains unchanged*     ; merging-masking
    FI;
ENDFOR
```

VPMOVSQB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 8
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SaturateSignedQuadWordToByte (SRC[m+63:m])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
```

```
        FI;
    ENDFOR
    DEST[MAXVL-1:VL/8] := 0;
```

VPMOVSQB instruction (EVEX encoded versions) when dest is memory

```
    (KL, VL) = (2, 128), (4, 256), (8, 512)
    FOR j := 0 TO KL-1
        i := j * 8
        m := j * 64
        IF k1[j] OR *no writemask*
            THEN DEST[i+7:i] := SaturateSignedQuadWordToByte (SRC[m+63:m])
            ELSE
                *DEST[i+7:i] remains unchanged*             ; merging-masking
        FI;
    ENDFOR
```

VPMOVUSQB instruction (EVEX encoded versions) when dest is a register

```
    (KL, VL) = (2, 128), (4, 256), (8, 512)
    FOR j := 0 TO KL-1
        i := j * 8
        m := j * 64
        IF k1[j] OR *no writemask*
            THEN DEST[i+7:i] := SaturateUnsignedQuadWordToByte (SRC[m+63:m])
            ELSE
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+7:i] remains unchanged*
                    ELSE *zeroing-masking*              ; zeroing-masking
                        DEST[i+7:i] := 0
                FI
        FI;
    ENDFOR
    DEST[MAXVL-1:VL/8] := 0;
```

VPMOVUSQB instruction (EVEX encoded versions) when dest is memory

```
    (KL, VL) = (2, 128), (4, 256), (8, 512)
    FOR j := 0 TO KL-1
        i := j * 8
        m := j * 64
        IF k1[j] OR *no writemask*
            THEN DEST[i+7:i] := SaturateUnsignedQuadWordToByte (SRC[m+63:m])
            ELSE
                *DEST[i+7:i] remains unchanged*             ; merging-masking
        FI;
    ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_mask_cvtepi64_epi8(__m128i s, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVSQB __m128i _mm512_cvtsepi64_epi8(__m512i a);
VPMOVSQB __m128i _mm512_mask_cvtsepi64_epi8(__m128i s, __mmask8 k, __m512i a);
VPMOVSQB __m128i _mm512_maskz_cvtsepi64_epi8(__mmask8 k, __m512i a);
VPMOVSQB void _mm512_mask_cvtsepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVUSQB __m128i _mm512_cvtusepi64_epi8(__m512i a);
VPMOVUSQB __m128i _mm512_mask_cvtusepi64_epi8(__m128i s, __mmask8 k, __m512i a);
VPMOVUSQB __m128i _mm512_maskz_cvtusepi64_epi8(__mmask8 k, __m512i a);
VPMOVUSQB void _mm512_mask_cvtusepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVUSQB __m128i _mm256_cvtusepi64_epi8(__m256i a);
VPMOVUSQB __m128i _mm256_mask_cvtusepi64_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVUSQB __m128i _mm256_maskz_cvtusepi64_epi8(__mmask8 k, __m256i b);
VPMOVUSQB void _mm256_mask_cvtusepi64_storeu_epi8(void * , __mmask8 k, __m256i b);
VPMOVUSQB __m128i _mm_cvtusepi64_epi8(__m128i a);
VPMOVUSQB __m128i _mm_mask_cvtusepi64_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVUSQB __m128i _mm_maskz_cvtusepi64_epi8(__mmask8 k, __m128i b);
VPMOVUSQB void _mm_mask_cvtusepi64_storeu_epi8(void * , __mmask8 k, __m128i b);
VPMOVSQB __m128i _mm256_cvtsepi64_epi8(__m256i a);
VPMOVSQB __m128i _mm256_mask_cvtsepi64_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVSQB __m128i _mm256_maskz_cvtsepi64_epi8(__mmask8 k, __m256i b);
VPMOVSQB void _mm256_mask_cvtsepi64_storeu_epi8(void * , __mmask8 k, __m256i b);
VPMOVSQB __m128i _mm_cvtsepi64_epi8(__m128i a);
VPMOVSQB __m128i _mm_mask_cvtsepi64_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVSQB __m128i _mm_maskz_cvtsepi64_epi8(__mmask8 k, __m128i b);
VPMOVSQB void _mm_mask_cvtsepi64_storeu_epi8(void * , __mmask8 k, __m128i b);
VPMOVQB __m128i _mm256_cvtepi64_epi8(__m256i a);
VPMOVQB __m128i _mm256_mask_cvtepi64_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVQB __m128i _mm256_maskz_cvtepi64_epi8(__mmask8 k, __m256i b);
VPMOVQB void _mm256_mask_cvtepi64_storeu_epi8(void * , __mmask8 k, __m256i b);
VPMOVQB __m128i _mm_cvtepi64_epi8(__m128i a);
VPMOVQB __m128i _mm_mask_cvtepi64_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVQB __m128i _mm_maskz_cvtepi64_epi8(__mmask8 k, __m128i b);
VPMOVQB void _mm_mask_cvtepi64_storeu_epi8(void * , __mmask8 k, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-55, "Type E6 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VPMOVQD/VPMOVSQD/VPMOVUSQD—Down Convert QWord to DWord

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 35 /r VPMOVQD xmm1/m128 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 2 packed quad-word integers from xmm2 into 2 packed double-word integers in xmm1/m128 with truncation subject to writemask k1.
EVEX.128.F3.0F38.W0 25 /r VPMOVSQD xmm1/m64 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 2 packed signed quad-word integers from xmm2 into 2 packed signed double-word integers in xmm1/m64 using signed saturation subject to writemask k1.
EVEX.128.F3.0F38.W0 15 /r VPMOVUSQD xmm1/m64 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 2 packed unsigned quad-word integers from xmm2 into 2 packed unsigned double-word integers in xmm1/m64 using unsigned saturation subject to writemask k1.
EVEX.256.F3.0F38.W0 35 /r VPMOVQD xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed quad-word integers from ymm2 into 4 packed double-word integers in xmm1/m128 with truncation subject to writemask k1.
EVEX.256.F3.0F38.W0 25 /r VPMOVSQD xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed signed quad-word integers from ymm2 into 4 packed signed double-word integers in xmm1/m128 using signed saturation subject to writemask k1.
EVEX.256.F3.0F38.W0 15 /r VPMOVUSQD xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed unsigned quad-word integers from ymm2 into 4 packed unsigned double-word integers in xmm1/m128 using unsigned saturation subject to writemask k1.
EVEX.512.F3.0F38.W0 35 /r VPMOVQD ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 8 packed quad-word integers from zmm2 into 8 packed double-word integers in ymm1/m256 with truncation subject to writemask k1.
EVEX.512.F3.0F38.W0 25 /r VPMOVSQD ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 8 packed signed quad-word integers from zmm2 into 8 packed signed double-word integers in ymm1/m256 using signed saturation subject to writemask k1.
EVEX.512.F3.0F38.W0 15 /r VPMOVUSQD ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 8 packed unsigned quad-word integers from zmm2 into 8 packed unsigned double-word integers in ymm1/m256 using unsigned saturation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

VPMOVQW down converts 64-bit integer elements in the source operand (the second operand) into packed double-words using truncation. VPMOVSQW converts signed 64-bit integers into packed signed doublewords using signed saturation. VPMOVUSQW convert unsigned quad-word values into unsigned double-word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted doubleword elements are written to the destination operand (the first operand) from the least-significant doubleword. Doubleword elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVQD instruction (EVEX encoded version) reg-reg form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TruncateQuadWordToDWord (SRC[m+63:m])
        ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0;
```

VPMOVQD instruction (EVEX encoded version) memory form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TruncateQuadWordToDWord (SRC[m+63:m])
        ELSE *DEST[i+31:i] remains unchanged*       ; merging-masking
    FI;
ENDFOR
```

VPMOVSQD instruction (EVEX encoded version) reg-reg form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SaturateSignedQuadWordToDWord (SRC[m+63:m])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
```

```
DEST[MAXVL-1:VL/2] := 0;
```

VPMOVSQD instruction (EVEX encoded version) memory form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SaturateSignedQuadWordToDWord (SRC[m+63:m])
        ELSE *DEST[i+31:i] remains unchanged*          ; merging-masking
    FI;
ENDFOR
```

VPMOVUSQD instruction (EVEX encoded version) reg-reg form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SaturateUnsignedQuadWordToDWord (SRC[m+63:m])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0;
```

VPMOVUSQD instruction (EVEX encoded version) memory form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SaturateUnsignedQuadWordToDWord (SRC[m+63:m])
        ELSE *DEST[i+31:i] remains unchanged*          ; merging-masking
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMOVQD __m256i _mm512_cvtepi64_epi32(__m512i a);
VPMOVQD __m256i _mm512_mask_cvtepi64_epi32(__m256i s, __mmask8 k, __m512i a);
VPMOVQD __m256i _mm512_maskz_cvtepi64_epi32(__mmask8 k, __m512i a);
VPMOVQD void _mm512_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVSQD __m256i _mm512_cvtsepi64_epi32(__m512i a);
VPMOVSQD __m256i _mm512_mask_cvtsepi64_epi32(__m256i s, __mmask8 k, __m512i a);
VPMOVSQD __m256i _mm512_maskz_cvtsepi64_epi32(__mmask8 k, __m512i a);
VPMOVSQD void _mm512_mask_cvtsepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVUSQD __m256i _mm512_cvtusepi64_epi32(__m512i a);
VPMOVUSQD __m256i _mm512_mask_cvtusepi64_epi32(__m256i s, __mmask8 k, __m512i a);
VPMOVUSQD __m256i _mm512_maskz_cvtusepi64_epi32(__mmask8 k, __m512i a);
VPMOVUSQD void _mm512_mask_cvtusepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVUSQD __m128i _mm256_cvtusepi64_epi32(__m256i a);
VPMOVUSQD __m128i _mm256_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVUSQD __m128i _mm256_maskz_cvtusepi64_epi32(__mmask8 k, __m256i b);
VPMOVUSQD void _mm256_mask_cvtusepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVUSQD __m128i _mm_cvtusepi64_epi32(__m128i a);
VPMOVUSQD __m128i _mm_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVUSQD __m128i _mm_maskz_cvtusepi64_epi32(__mmask8 k, __m128i b);
VPMOVUSQD void _mm_mask_cvtusepi64_storeu_epi32(void * , __mmask8 k, __m128i b);
VPMOVSQD __m128i _mm256_cvtsepi64_epi32(__m256i a);
VPMOVSQD __m128i _mm256_mask_cvtsepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVSQD __m128i _mm256_maskz_cvtsepi64_epi32(__mmask8 k, __m256i b);
VPMOVSQD void _mm256_mask_cvtsepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVSQD __m128i _mm_cvtsepi64_epi32(__m128i a);
VPMOVSQD __m128i _mm_mask_cvtsepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVSQD __m128i _mm_maskz_cvtsepi64_epi32(__mmask8 k, __m128i b);
VPMOVSQD void _mm_mask_cvtsepi64_storeu_epi32(void * , __mmask8 k, __m128i b);
VPMOVQD __m128i _mm256_cvtepi64_epi32(__m256i a);
VPMOVQD __m128i _mm256_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVQD __m128i _mm256_maskz_cvtepi64_epi32(__mmask8 k, __m256i b);
VPMOVQD void _mm256_mask_cvtepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVQD __m128i _mm_cvtepi64_epi32(__m128i a);
VPMOVQD __m128i _mm_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVQD __m128i _mm_maskz_cvtepi64_epi32(__mmask8 k, __m128i b);
VPMOVQD void _mm_mask_cvtepi64_storeu_epi32(void * , __mmask8 k, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-55, "Type E6 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VPMOVQW/VPMOVSQW/VPMOVUSQW—Down Convert QWord to Word

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 34 /r VPMOVQW xmm1/m32 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 2 packed quad-word integers from xmm2 into 2 packed word integers in xmm1/m32 with truncation under writemask k1.
EVEX.128.F3.0F38.W0 24 /r VPMOVSQW xmm1/m32 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 8 packed signed quad-word integers from zmm2 into 8 packed signed word integers in xmm1/m32 using signed saturation under writemask k1.
EVEX.128.F3.0F38.W0 14 /r VPMOVUSQW xmm1/m32 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 2 packed unsigned quad-word integers from xmm2 into 2 packed unsigned word integers in xmm1/m32 using unsigned saturation under writemask k1.
EVEX.256.F3.0F38.W0 34 /r VPMOVQW xmm1/m64 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed quad-word integers from ymm2 into 4 packed word integers in xmm1/m64 with truncation under writemask k1.
EVEX.256.F3.0F38.W0 24 /r VPMOVSQW xmm1/m64 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed signed quad-word integers from ymm2 into 4 packed signed word integers in xmm1/m64 using signed saturation under writemask k1.
EVEX.256.F3.0F38.W0 14 /r VPMOVUSQW xmm1/m64 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed unsigned quad-word integers from ymm2 into 4 packed unsigned word integers in xmm1/m64 using unsigned saturation under writemask k1.
EVEX.512.F3.0F38.W0 34 /r VPMOVQW xmm1/m128 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 8 packed quad-word integers from zmm2 into 8 packed word integers in xmm1/m128 with truncation under writemask k1.
EVEX.512.F3.0F38.W0 24 /r VPMOVSQW xmm1/m128 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 8 packed signed quad-word integers from zmm2 into 8 packed signed word integers in xmm1/m128 using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 14 /r VPMOVUSQW xmm1/m128 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 8 packed unsigned quad-word integers from zmm2 into 8 packed unsigned word integers in xmm1/m128 using unsigned saturation under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Quarter Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

VPMOVQW down converts 64-bit integer elements in the source operand (the second operand) into packed words using truncation. VPMOVSQW converts signed 64-bit integers into packed signed words using signed saturation. VPMOVUSQW convert unsigned quad-word values into unsigned word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a XMM register or a 128/64/32-bit memory location.

Down-converted word elements are written to the destination operand (the first operand) from the least-significant word. Word elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:128/64/32) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVQW instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 16
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TruncateQuadWordToWord (SRC[m+63:m])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/4] := 0;
```

VPMOVQW instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 16
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TruncateQuadWordToWord (SRC[m+63:m])
        ELSE
            *DEST[i+15:i] remains unchanged*    ; merging-masking
    FI;
ENDFOR
```

VPMOVSQW instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 16
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SaturateSignedQuadWordToWord (SRC[m+63:m])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] := 0
```

```
                FI
        FI;
    ENDFOR
    DEST[MAXVL-1:VL/4] := 0;
```

VPMOVSQW instruction (EVEX encoded versions) when dest is memory

```
    (KL, VL) = (2, 128), (4, 256), (8, 512)
    FOR j := 0 TO KL-1
        i := j * 16
        m := j * 64
        IF k1[j] OR *no writemask*
            THEN DEST[i+15:i] := SaturateSignedQuadWordToWord (SRC[m+63:m])
            ELSE
                *DEST[i+15:i] remains unchanged*     ; merging-masking
        FI;
    ENDFOR
```

VPMOVUSQW instruction (EVEX encoded versions) when dest is a register

```
    (KL, VL) = (2, 128), (4, 256), (8, 512)
    FOR j := 0 TO KL-1
        i := j * 16
        m := j * 64
        IF k1[j] OR *no writemask*
            THEN DEST[i+15:i] := SaturateUnsignedQuadWordToWord (SRC[m+63:m])
            ELSE
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+15:i] remains unchanged*
                    ELSE *zeroing-masking*              ; zeroing-masking
                        DEST[i+15:i] := 0
                FI
        FI;
    ENDFOR
    DEST[MAXVL-1:VL/4] := 0;
```

VPMOVUSQW instruction (EVEX encoded versions) when dest is memory

```
    (KL, VL) = (2, 128), (4, 256), (8, 512)
    FOR j := 0 TO KL-1
        i := j * 16
        m := j * 64
        IF k1[j] OR *no writemask*
            THEN DEST[i+15:i] := SaturateUnsignedQuadWordToWord (SRC[m+63:m])
            ELSE
                *DEST[i+15:i] remains unchanged*     ; merging-masking
        FI;
    ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMOVQW __m128i _mm512_cvtepi64_epi16(__m512i a);
VPMOVQW __m128i _mm512_mask_cvtepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVQW __m128i _mm512_maskz_cvtepi64_epi16(__mmask8 k, __m512i a);
VPMOVQW void _mm512_mask_cvtepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);
VPMOVSQW __m128i _mm512_cvtsepi64_epi16(__m512i a);
VPMOVSQW __m128i _mm512_mask_cvtsepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVSQW __m128i _mm512_maskz_cvtsepi64_epi16(__mmask8 k, __m512i a);
VPMOVSQW void _mm512_mask_cvtsepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);
VPMOVUSQW __m128i _mm512_cvtusepi64_epi16(__m512i a);
VPMOVUSQW __m128i _mm512_mask_cvtusepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVUSQW __m128i _mm512_maskz_cvtusepi64_epi16(__mmask8 k, __m512i a);
VPMOVUSQW void _mm512_mask_cvtusepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);
VPMOVUSQD __m128i _mm256_cvtusepi64_epi32(__m256i a);
VPMOVUSQD __m128i _mm256_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVUSQD __m128i _mm256_maskz_cvtusepi64_epi32(__mmask8 k, __m256i b);
VPMOVUSQD void _mm256_mask_cvtusepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVUSQD __m128i _mm_cvtusepi64_epi32(__m128i a);
VPMOVUSQD __m128i _mm_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVUSQD __m128i _mm_maskz_cvtusepi64_epi32(__mmask8 k, __m128i b);
VPMOVUSQD void _mm_mask_cvtusepi64_storeu_epi32(void * , __mmask8 k, __m128i b);
VPMOVSQD __m128i _mm256_cvtsepi64_epi32(__m256i a);
VPMOVSQD __m128i _mm256_mask_cvtsepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVSQD __m128i _mm256_maskz_cvtsepi64_epi32(__mmask8 k, __m256i b);
VPMOVSQD void _mm256_mask_cvtsepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVSQD __m128i _mm_cvtsepi64_epi32(__m128i a);
VPMOVSQD __m128i _mm_mask_cvtsepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVSQD __m128i _mm_maskz_cvtsepi64_epi32(__mmask8 k, __m128i b);
VPMOVSQD void _mm_mask_cvtsepi64_storeu_epi32(void * , __mmask8 k, __m128i b);
VPMOVQD __m128i _mm256_cvtepi64_epi32(__m256i a);
VPMOVQD __m128i _mm256_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVQD __m128i _mm256_maskz_cvtepi64_epi32(__mmask8 k, __m256i b);
VPMOVQD void _mm256_mask_cvtepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVQD __m128i _mm_cvtepi64_epi32(__m128i a);
VPMOVQD __m128i _mm_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVQD __m128i _mm_maskz_cvtepi64_epi32(__mmask8 k, __m128i b);
VPMOVQD void _mm_mask_cvtepi64_storeu_epi32(void * , __mmask8 k, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-55, "Type E6 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VPMOVWB/VPMOVSWB/VPMOVUSWB—Down Convert Word to Byte

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 30 /r VPMOVWB xmm1/m64 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Converts 8 packed word integers from xmm2 into 8 packed bytes in xmm1/m64 with truncation under writemask k1.
EVEX.128.F3.0F38.W0 20 /r VPMOVSWB xmm1/m64 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Converts 8 packed signed word integers from xmm2 into 8 packed signed bytes in xmm1/m64 using signed saturation under writemask k1.
EVEX.128.F3.0F38.W0 10 /r VPMOVUSWB xmm1/m64 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Converts 8 packed unsigned word integers from xmm2 into 8 packed unsigned bytes in 8mm1/m64 using unsigned saturation under writemask k1.
EVEX.256.F3.0F38.W0 30 /r VPMOVWB xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Converts 16 packed word integers from ymm2 into 16 packed bytes in xmm1/m128 with truncation under writemask k1.
EVEX.256.F3.0F38.W0 20 /r VPMOVSWB xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Converts 16 packed signed word integers from ymm2 into 16 packed signed bytes in xmm1/m128 using signed saturation under writemask k1.
EVEX.256.F3.0F38.W0 10 /r VPMOVUSWB xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Converts 16 packed unsigned word integers from ymm2 into 16 packed unsigned bytes in xmm1/m128 using unsigned saturation under writemask k1.
EVEX.512.F3.0F38.W0 30 /r VPMOVWB ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512BW OR AVX10.1	Converts 32 packed word integers from zmm2 into 32 packed bytes in ymm1/m256 with truncation under writemask k1.
EVEX.512.F3.0F38.W0 20 /r VPMOVSWB ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512BW OR AVX10.1	Converts 32 packed signed word integers from zmm2 into 32 packed signed bytes in ymm1/m256 using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 10 /r VPMOVUSWB ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512BW OR AVX10.1	Converts 32 packed unsigned word integers from zmm2 into 32 packed unsigned bytes in ymm1/m256 using unsigned saturation under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

VPMOVWB down converts 16-bit integers into packed bytes using truncation. VPMOVSWB converts signed 16-bit integers into packed signed bytes using signed saturation. VPMOVUSWB convert unsigned word values into unsigned byte values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVWB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 8
    m := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TruncateWordToByte (SRC[m+15:m])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0;
```

VPMOVWB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 8
    m := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TruncateWordToByte (SRC[m+15:m])
        ELSE
            *DEST[i+7:i] remains unchanged*     ; merging-masking
    FI;
ENDFOR
```

VPMOVSWB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 8
    m := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SaturateSignedWordToByte (SRC[m+15:m])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0;
```

VPMOVSWB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KI-1
    i := j * 8
    m := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SaturateSignedWordToByte (SRC[m+15:m])
        ELSE
            *DEST[i+7:i] remains unchanged*      ; merging-masking
    FI;
ENDFOR
```

VPMOVUSWB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KI-1
    i := j * 8
    m := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SaturateUnsignedWordToByte (SRC[m+15:m])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0;
```

VPMOVUSWB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KI-1
    i := j * 8
    m := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SaturateUnsignedWordToByte (SRC[m+15:m])
        ELSE
            *DEST[i+7:i] remains unchanged*      ; merging-masking
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMOVUSWB __m256i _mm512_cvtusepi16_epi8(__m512i a);
VPMOVUSWB __m256i _mm512_mask_cvtusepi16_epi8(__m256i a, __mmask32 k, __m512i b);
VPMOVUSWB __m256i _mm512_maskz_cvtusepi16_epi8(__mmask32 k, __m512i b);
VPMOVUSWB void _mm512_mask_cvtusepi16_storeu_epi8(void * , __mmask32 k, __m512i b);
VPMOVSWB __m256i _mm512_cvtsepi16_epi8(__m512i a);
VPMOVSWB __m256i _mm512_mask_cvtsepi16_epi8(__m256i a, __mmask32 k, __m512i b);
VPMOVSWB __m256i _mm512_maskz_cvtsepi16_epi8(__mmask32 k, __m512i b);
VPMOVSWB void _mm512_mask_cvtsepi16_storeu_epi8(void * , __mmask32 k, __m512i b);
VPMOVWB __m256i _mm512_cvtepi16_epi8(__m512i a);
VPMOVWB __m256i _mm512_mask_cvtepi16_epi8(__m256i a, __mmask32 k, __m512i b);
VPMOVWB __m256i _mm512_maskz_cvtepi16_epi8(__mmask32 k, __m512i b);
VPMOVWB void _mm512_mask_cvtepi16_storeu_epi8(void * , __mmask32 k, __m512i b);
VPMOVUSWB __m128i _mm256_cvtusepi16_epi8(__m256i a);
VPMOVUSWB __m128i _mm256_mask_cvtusepi16_epi8(__m128i a, __mmask16 k, __m256i b);
VPMOVUSWB __m128i _mm256_maskz_cvtusepi16_epi8(__mmask16 k, __m256i b);
VPMOVUSWB void _mm256_mask_cvtusepi16_storeu_epi8(void * , __mmask16 k, __m256i b);
VPMOVUSWB __m128i _mm_cvtusepi16_epi8(__m128i a);
VPMOVUSWB __m128i _mm_mask_cvtusepi16_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVUSWB __m128i _mm_maskz_cvtusepi16_epi8(__mmask8 k, __m128i b);
VPMOVUSWB void _mm_mask_cvtusepi16_storeu_epi8(void * , __mmask8 k, __m128i b);
VPMOVSWB __m128i _mm256_cvtsepi16_epi8(__m256i a);
VPMOVSWB __m128i _mm256_mask_cvtsepi16_epi8(__m128i a, __mmask16 k, __m256i b);
VPMOVSWB __m128i _mm256_maskz_cvtsepi16_epi8(__mmask16 k, __m256i b);
VPMOVSWB void _mm256_mask_cvtsepi16_storeu_epi8(void * , __mmask16 k, __m256i b);
VPMOVSWB __m128i _mm_cvtsepi16_epi8(__m128i a);
VPMOVSWB __m128i _mm_mask_cvtsepi16_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVSWB __m128i _mm_maskz_cvtsepi16_epi8(__mmask8 k, __m128i b);
VPMOVSWB void _mm_mask_cvtsepi16_storeu_epi8(void * , __mmask8 k, __m128i b);
VPMOVWB __m128i _mm256_cvtepi16_epi8(__m256i a);
VPMOVWB __m128i _mm256_mask_cvtepi16_epi8(__m128i a, __mmask16 k, __m256i b);
VPMOVWB __m128i _mm256_maskz_cvtepi16_epi8(__mmask16 k, __m256i b);
VPMOVWB void _mm256_mask_cvtepi16_storeu_epi8(void * , __mmask16 k, __m256i b);
VPMOVWB __m128i _mm_cvtepi16_epi8(__m128i a);
VPMOVWB __m128i _mm_mask_cvtepi16_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVWB __m128i _mm_maskz_cvtepi16_epi8(__mmask8 k, __m128i b);
VPMOVWB void _mm_mask_cvtepi16_storeu_epi8(void * , __mmask8 k, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-55, "Type E6 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VPMULTISHIFTQB—Select Packed Unaligned Bytes From Quadword Sources

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 83 /r VPMULTISHIFTQB xmm1 {k1}{z}, xmm2,xmm3/m128/m64bcst	A	V/V	(AVX512VL AND AVX512_VBMI) OR AVX10.1	Select unaligned bytes from qwords in xmm3/m128/m64bcst using control bytes in xmm2, write byte results to xmm1 under k1.
EVEX.256.66.0F38.W1 83 /r VPMULTISHIFTQB ymm1 {k1}{z}, ymm2,ymm3/m256/m64bcst	A	V/V	(AVX512VL AVX512_VBMI) OR AVX10.1	Select unaligned bytes from qwords in ymm3/m256/m64bcst using control bytes in ymm2, write byte results to ymm1 under k1.
EVEX.512.66.0F38.W1 83 /r VPMULTISHIFTQB zmm1 {k1}{z}, zmm2,zmm3/m512/m64bcst	A	V/V	AVX512_VBMI OR AVX10.1	Select unaligned bytes from qwords in zmm3/m512/m64bcst using control bytes in zmm2, write byte results to zmm1 under k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction selects eight unaligned bytes from each input qword element of the second source operand (the third operand) and writes eight assembled bytes for each qword element in the destination operand (the first operand). Each byte result is selected using a byte-granular shift control within the corresponding qword element of the first source operand (the second operand). Each byte result in the destination operand is updated under the writemask k1.

Only the low 6 bits of each control byte are used to select an 8-bit slot to extract the output byte from the qword data in the second source operand. The starting bit of the 8-bit slot can be unaligned relative to any byte boundary and is extracted from the input qword source at the location specified in the low 6-bit of the control byte. If the 8-bit slot would exceed the qword boundary, the out-of-bound portion of the 8-bit slot is wrapped back to start from bit 0 of the input qword element.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register.

Operation

VPMULTISHIFTQB DEST, SRC1, SRC2 (EVEX encoded version)

```
(KL, VL) = (2, 128),(4, 256), (8, 512)
FOR i := 0 TO KL-1
    IF EVEX.b=1 AND src2 is memory THEN
            tcur := src2.qword[0]; //broadcasting
    ELSE
            tcur := src2.qword[i];
    FI;
    FOR j := 0 to 7
        ctrl := src1.qword[i].byte[j] & 63;
        FOR k := 0 to 7
            res.bit[k] := tcur.bit[ (ctrl+k) mod 64 ];
        ENDFOR
        IF k1[i*8+j] or no writemask THEN
            DEST.qword[i].byte[j] := res;
        ELSE IF zeroing-masking THEN
            DEST.qword[i].byte[j] := 0;
    ENDFOR
ENDFOR
DEST.qword[MAX_VL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMULTISHIFTQB __m512i _mm512_multishift_epi64_epi8(__m512i a, __m512i b);
VPMULTISHIFTQB __m512i _mm512_mask_multishift_epi64_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMULTISHIFTQB __m512i _mm512_maskz_multishift_epi64_epi8(__mmask64 k, __m512i a, __m512i b);
VPMULTISHIFTQB __m256i _mm256_multishift_epi64_epi8(__m256i a, __m256i b);
VPMULTISHIFTQB __m256i _mm256_mask_multishift_epi64_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMULTISHIFTQB __m256i _mm256_maskz_multishift_epi64_epi8(__mmask32 k, __m256i a, __m256i b);
VPMULTISHIFTQB __m128i _mm_multishift_epi64_epi8(__m128i a, __m128i b);
VPMULTISHIFTQB __m128i _mm_mask_multishift_epi64_epi8(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULTISHIFTQB __m128i _mm_maskz_multishift_epi64_epi8(__mmask8 k, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-52, "Type E4NF Class Exception Conditions."

VPOPCNT—Return the Count of Number of Bits Set to 1 in BYTE/WORD/DWORD/QWORD

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 54 /r VPOPCNTB xmm1{k1}{z}, xmm2/m128	A	V/V	(AVX512_BITALG AND AVX512VL) OR AVX10.1	Counts the number of bits set to one in xmm2/m128 and puts the result in xmm1 with writemask k1.
EVEX.256.66.0F38.W0 54 /r VPOPCNTB ymm1{k1}{z}, ymm2/m256	A	V/V	(AVX512_BITALG AND AVX512VL) OR AVX10.1	Counts the number of bits set to one in ymm2/m256 and puts the result in ymm1 with writemask k1.
EVEX.512.66.0F38.W0 54 /r VPOPCNTB zmm1{k1}{z}, zmm2/m512	A	V/V	AVX512_BITALG OR AVX10.1	Counts the number of bits set to one in zmm2/m512 and puts the result in zmm1 with writemask k1.
EVEX.128.66.0F38.W1 54 /r VPOPCNTW xmm1{k1}{z}, xmm2/m128	A	V/V	(AVX512_BITALG AND AVX512VL) OR AVX10.1	Counts the number of bits set to one in xmm2/m128 and puts the result in xmm1 with writemask k1.
EVEX.256.66.0F38.W1 54 /r VPOPCNTW ymm1{k1}{z}, ymm2/m256	A	V/V	(AVX512_BITALG AND AVX512VL) OR AVX10.1	Counts the number of bits set to one in ymm2/m256 and puts the result in ymm1 with writemask k1.
EVEX.512.66.0F38.W1 54 /r VPOPCNTW zmm1{k1}{z}, zmm2/m512	A	V/V	AVX512_BITALG OR AVX10.1	Counts the number of bits set to one in zmm2/m512 and puts the result in zmm1 with writemask k1.
EVEX.128.66.0F38.W0 55 /r VPOPCNTD xmm1{k1}{z}, xmm2/m128/m32bcst	B	V/V	(AVX512_VPOPCNTDQ AND AVX512VL) OR AVX10.1	Counts the number of bits set to one in xmm2/m128/m32bcst and puts the result in xmm1 with writemask k1.
EVEX.256.66.0F38.W0 55 /r VPOPCNTD ymm1{k1}{z}, ymm2/m256/m32bcst	B	V/V	(AVX512_VPOPCNTDQ AND AVX512VL) OR AVX10.1	Counts the number of bits set to one in ymm2/m256/m32bcst and puts the result in ymm1 with writemask k1.
EVEX.512.66.0F38.W0 55 /r VPOPCNTD zmm1{k1}{z}, zmm2/m512/m32bcst	B	V/V	AVX512_VPOPCNTDQ OR AVX10.1	Counts the number of bits set to one in zmm2/m512/m32bcst and puts the result in zmm1 with writemask k1.
EVEX.128.66.0F38.W1 55 /r VPOPCNTQ xmm1{k1}{z}, xmm2/m128/m64bcst	B	V/V	(AVX512_VPOPCNTDQ AND AVX512VL) OR AVX10.1	Counts the number of bits set to one in xmm2/m128/m32bcst and puts the result in xmm1 with writemask k1.
EVEX.256.66.0F38.W1 55 /r VPOPCNTQ ymm1{k1}{z}, ymm2/m256/m64bcst	B	V/V	(AVX512_VPOPCNTDQ AND AVX512VL) OR AVX10.1	Counts the number of bits set to one in ymm2/m256/m32bcst and puts the result in ymm1 with writemask k1.
EVEX.512.66.0F38.W1 55 /r VPOPCNTQ zmm1{k1}{z}, zmm2/m512/m64bcst	B	V/V	AVX512_VPOPCNTDQ OR AVX10.1	Counts the number of bits set to one in zmm2/m512/m64bcst and puts the result in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction counts the number of bits set to one in each byte, word, dword or qword element of its source (e.g., zmm2 or memory) and places the results in the destination register (zmm1). This instruction supports memory fault suppression.

Operation

VPOPCNTB

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        DEST.byte[j] := POPCNT(SRC.byte[j])
    ELSE IF *merging-masking*:
        *DEST.byte[j] remains unchanged*
    ELSE:
        DEST.byte[j] := 0
DEST[MAX_VL-1:VL] := 0
```

VPOPCNTW

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        DEST.word[j] := POPCNT(SRC.word[j])
    ELSE IF *merging-masking*:
        *DEST.word[j] remains unchanged*
    ELSE:
        DEST.word[j] := 0
DEST[MAX_VL-1:VL] := 0
```

VPOPCNTD

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        IF SRC is broadcast memop:
            t := SRC.dword[0]
        ELSE:
            t := SRC.dword[j]
        DEST.dword[j] := POPCNT(t)
    ELSE IF *merging-masking*:
        *DEST..dword[j] remains unchanged*
    ELSE:
        DEST..dword[j] := 0
DEST[MAX_VL-1:VL] := 0
```

VPOPCNTQ

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        IF SRC is broadcast memop:
            t := SRC.qword[0]
        ELSE:
            t := SRC.qword[j]
        DEST.qword[j] := POPCNT(t)
    ELSE IF *merging-masking*:
        *DEST..qword[j] remains unchanged*
    ELSE:
        DEST..qword[j] := 0
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPOPCNTW __m128i _mm_popcnt_epi16(__m128i);
VPOPCNTW __m128i _mm_mask_popcnt_epi16(__m128i, __mmask8, __m128i);
VPOPCNTW __m128i _mm_maskz_popcnt_epi16(__mmask8, __m128i);
VPOPCNTW __m256i _mm256_popcnt_epi16(__m256i);
VPOPCNTW __m256i _mm256_mask_popcnt_epi16(__m256i, __mmask16, __m256i);
VPOPCNTW __m256i _mm256_maskz_popcnt_epi16(__mmask16, __m256i);
VPOPCNTW __m512i _mm512_popcnt_epi16(__m512i);
VPOPCNTW __m512i _mm512_mask_popcnt_epi16(__m512i, __mmask32, __m512i);
VPOPCNTW __m512i  _mm512_maskz_popcnt_epi16(__mmask32, __m512i);
VPOPCNTQ __m128i  _mm_popcnt_epi64(__m128i);
VPOPCNTQ __m128i  _mm_mask_popcnt_epi64(__m128i, __mmask8, __m128i);
VPOPCNTQ __m128i  _mm_maskz_popcnt_epi64(__mmask8, __m128i);
VPOPCNTQ __m256i  _mm256_popcnt_epi64(__m256i);
VPOPCNTQ __m256i  _mm256_mask_popcnt_epi64(__m256i, __mmask8, __m256i);
VPOPCNTQ __m256i  _mm256_maskz_popcnt_epi64(__mmask8, __m256i);
VPOPCNTQ __m512i  _mm512_popcnt_epi64(__m512i);
VPOPCNTQ __m512i  _mm512_mask_popcnt_epi64(__m512i, __mmask8, __m512i);
VPOPCNTQ __m512i  _mm512_maskz_popcnt_epi64(__mmask8, __m512i);
VPOPCNTD __m128i  _mm_popcnt_epi32(__m128i);
VPOPCNTD __m128i  _mm_mask_popcnt_epi32(__m128i, __mmask8, __m128i);
VPOPCNTD __m128i  _mm_maskz_popcnt_epi32(__mmask8, __m128i);
VPOPCNTD __m256i  _mm256_popcnt_epi32(__m256i);
VPOPCNTD __m256i  _mm256_mask_popcnt_epi32(__m256i, __mmask8, __m256i);
VPOPCNTD __m256i  _mm256_maskz_popcnt_epi32(__mmask8, __m256i);
VPOPCNTD __m512i  _mm512_popcnt_epi32(__m512i);
VPOPCNTD __m512i  _mm512_mask_popcnt_epi32(__m512i, __mmask16, __m512i);
VPOPCNTD __m512i  _mm512_maskz_popcnt_epi32(__mmask16, __m512i);
VPOPCNTB __m128i  _mm_popcnt_epi8(__m128i);
VPOPCNTB __m128i  _mm_mask_popcnt_epi8(__m128i, __mmask16, __m128i);
VPOPCNTB __m128i  _mm_maskz_popcnt_epi8(__mmask16, __m128i);
VPOPCNTB __m256i  _mm256_popcnt_epi8(__m256i);
VPOPCNTB __m256i  _mm256_mask_popcnt_epi8(__m256i, __mmask32, __m256i);
VPOPCNTB __m256i  _mm256_maskz_popcnt_epi8(__mmask32, __m256i);
VPOPCNTB __m512i  _mm512_popcnt_epi8(__m512i);
VPOPCNTB __m512i  _mm512_mask_popcnt_epi8(__m512i, __mmask64, __m512i);
VPOPCNTB __m512i  _mm512_maskz_popcnt_epi8(__mmask64, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, "Type E4 Class Exception Conditions."

VPROLD/VPROLVD/VPROLQ/VPROLVQ—Bit Rotate Left

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 15 /r VPROLVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate doublewords in xmm2 left by count in the corresponding element of xmm3/m128/m32bcst. Result written to xmm1 under writemask k1.
EVEX.128.66.0F.W0 72 /1 ib VPROLD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate doublewords in xmm2/m128/m32bcst left by imm8. Result written to xmm1 using writemask k1.
EVEX.128.66.0F38.W1 15 /r VPROLVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate quadwords in xmm2 left by count in the corresponding element of xmm3/m128/m64bcst. Result written to xmm1 under writemask k1.
EVEX.128.66.0F.W1 72 /1 ib VPROLQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate quadwords in xmm2/m128/m64bcst left by imm8. Result written to xmm1 using writemask k1.
EVEX.256.66.0F38.W0 15 /r VPROLVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate doublewords in ymm2 left by count in the corresponding element of ymm3/m256/m32bcst. Result written to ymm1 under writemask k1.
EVEX.256.66.0F.W0 72 /1 ib VPROLD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate doublewords in ymm2/m256/m32bcst left by imm8. Result written to ymm1 using writemask k1.
EVEX.256.66.0F38.W1 15 /r VPROLVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate quadwords in ymm2 left by count in the corresponding element of ymm3/m256/m64bcst. Result written to ymm1 under writemask k1.
EVEX.256.66.0F.W1 72 /1 ib VPROLQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate quadwords in ymm2/m256/m64bcst left by imm8. Result written to ymm1 using writemask k1.
EVEX.512.66.0F38.W0 15 /r VPROLVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F OR AVX10.1[1]	Rotate left of doublewords in zmm2 by count in the corresponding element of zmm3/m512/m32bcst. Result written to zmm1 using writemask k1.
EVEX.512.66.0F.W0 72 /1 ib VPROLD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	A	V/V	AVX512F OR AVX10.1	Rotate left of doublewords in zmm3/m512/m32bcst by imm8. Result written to zmm1 using writemask k1.
EVEX.512.66.0F38.W1 15 /r VPROLVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F OR AVX10.1	Rotate quadwords in zmm2 left by count in the corresponding element of zmm3/m512/m64bcst. Result written to zmm1under writemask k1.
EVEX.512.66.0F.W1 72 /1 ib VPROLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	A	V/V	AVX512F OR AVX10.1	Rotate quadwords in zmm2/m512/m64bcst left by imm8. Result written to zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	VEX.vvvv (w)	ModRM:r/m (R)	imm8	N/A
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Rotates the bits in the individual data elements (doublewords, or quadword) in the first source operand to the left by the number of bits specified in the count operand. If the value specified by the count operand is greater than 31 (for doublewords), or 63 (for a quadword), then the count operand modulo the data size (32 or 64) is used.

EVEX.128 encoded version: The destination operand is a XMM register. The source operand is a XMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

EVEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The destination operand is a ZMM register updated according to the writemask. For the count operand in immediate form, the source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location, the count operand is an 8-bit immediate. For the count operand in variable form, the first source operand (the second operand) is a ZMM register and the counter operand (the third operand) is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location.

Operation

```
LEFT_ROTATE_DWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC modulo 32;
DEST[31:0] := (SRC << COUNT) | (SRC >> (32 - COUNT));

LEFT_ROTATE_QWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC modulo 64;
DEST[63:0] := (SRC << COUNT) | (SRC >> (64 - COUNT));
```

VPROLD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+31:i] := LEFT_ROTATE_DWORDS(SRC1[31:0], imm8)
                ELSE DEST[i+31:i] := LEFT_ROTATE_DWORDS(SRC1[i+31:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPROLVD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := LEFT_ROTATE_DWORDS(SRC1[i+31:i], SRC2[31:0])
                ELSE DEST[i+31:i] := LEFT_ROTATE_DWORDS(SRC1[i+31:i], SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPROLQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+63:i] := LEFT_ROTATE_QWORDS(SRC1[63:0], imm8)
                ELSE DEST[i+63:i] := LEFT_ROTATE_QWORDS(SRC1[i+63:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPROLVQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := LEFT_ROTATE_QWORDS(SRC1[i+63:i], SRC2[63:0])
                ELSE DEST[i+63:i] := LEFT_ROTATE_QWORDS(SRC1[i+63:i], SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
```

```
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPROLD __m512i _mm512_rol_epi32(__m512i a, int imm);
VPROLD __m512i _mm512_mask_rol_epi32(__m512i a, __mmask16 k, __m512i b, int imm);
VPROLD __m512i _mm512_maskz_rol_epi32(__mmask16 k, __m512i a, int imm);
VPROLD __m256i _mm256_rol_epi32(__m256i a, int imm);
VPROLD __m256i _mm256_mask_rol_epi32(__m256i a, __mmask8 k, __m256i b, int imm);
VPROLD __m256i _mm256_maskz_rol_epi32(__mmask8 k, __m256i a, int imm);
VPROLD __m128i _mm_rol_epi32(__m128i a, int imm);
VPROLD __m128i _mm_mask_rol_epi32(__m128i a, __mmask8 k, __m128i b, int imm);
VPROLD __m128i _mm_maskz_rol_epi32(__mmask8 k, __m128i a, int imm);
VPROLQ __m512i _mm512_rol_epi64(__m512i a, int imm);
VPROLQ __m512i _mm512_mask_rol_epi64(__m512i a, __mmask8 k, __m512i b, int imm);
VPROLQ __m512i _mm512_maskz_rol_epi64(__mmask8 k, __m512i a, int imm);
VPROLQ __m256i _mm256_rol_epi64(__m256i a, int imm);
VPROLQ __m256i _mm256_mask_rol_epi64(__m256i a, __mmask8 k, __m256i b, int imm);
VPROLQ __m256i _mm256_maskz_rol_epi64(__mmask8 k, __m256i a, int imm);
VPROLQ __m128i _mm_rol_epi64(__m128i a, int imm);
VPROLQ __m128i _mm_mask_rol_epi64(__m128i a, __mmask8 k, __m128i b, int imm);
VPROLQ __m128i _mm_maskz_rol_epi64(__mmask8 k, __m128i a, int imm);
VPROLVD __m512i _mm512_rolv_epi32(__m512i a, __m512i cnt);
VPROLVD __m512i _mm512_mask_rolv_epi32(__m512i a, __mmask16 k, __m512i b, __m512i cnt);
VPROLVD __m512i _mm512_maskz_rolv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPROLVD __m256i _mm256_rolv_epi32(__m256i a, __m256i cnt);
VPROLVD __m256i _mm256_mask_rolv_epi32(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPROLVD __m256i _mm256_maskz_rolv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPROLVD __m128i _mm_rolv_epi32(__m128i a, __m128i cnt);
VPROLVD __m128i _mm_mask_rolv_epi32(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPROLVD __m128i _mm_maskz_rolv_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPROLVQ __m512i _mm512_rolv_epi64(__m512i a, __m512i cnt);
VPROLVQ __m512i _mm512_mask_rolv_epi64(__m512i a, __mmask8 k, __m512i b, __m512i cnt);
VPROLVQ __m512i _mm512_maskz_rolv_epi64(__mmask8 k, __m512i a, __m512i cnt);
VPROLVQ __m256i _mm256_rolv_epi64(__m256i a, __m256i cnt);
VPROLVQ __m256i _mm256_mask_rolv_epi64(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPROLVQ __m256i _mm256_maskz_rolv_epi64(__mmask8 k, __m256i a, __m256i cnt);
VPROLVQ __m128i _mm_rolv_epi64(__m128i a, __m128i cnt);
VPROLVQ __m128i _mm_mask_rolv_epi64(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPROLVQ __m128i _mm_maskz_rolv_epi64(__mmask8 k, __m128i a, __m128i cnt);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

VPRORD/VPRORVD/VPRORQ/VPRORVQ—Bit Rotate Right

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 14 /r VPRORVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate doublewords in xmm2 right by count in the corresponding element of xmm3/m128/m32bcst, store result using writemask k1.
EVEX.128.66.0F.W0 72 /0 ib VPRORD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate doublewords in xmm2/m128/m32bcst right by imm8, store result using writemask k1.
EVEX.128.66.0F38.W1 14 /r VPRORVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate quadwords in xmm2 right by count in the corresponding element of xmm3/m128/m64bcst, store result using writemask k1.
EVEX.128.66.0F.W1 72 /0 ib VPRORQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate quadwords in xmm2/m128/m64bcst right by imm8, store result using writemask k1.
EVEX.256.66.0F38.W0 14 /r VPRORVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate doublewords in ymm2 right by count in the corresponding element of ymm3/m256/m32bcst, store using result writemask k1.
EVEX.256.66.0F.W0 72 /0 ib VPRORD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate doublewords in ymm2/m256/m32bcst right by imm8, store result using writemask k1.
EVEX.256.66.0F38.W1 14 /r VPRORVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate quadwords in ymm2 right by count in the corresponding element of ymm3/m256/m64bcst, store result using writemask k1.
EVEX.256.66.0F.W1 72 /0 ib VPRORQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate quadwords in ymm2/m256/m64bcst right by imm8, store result using writemask k1.
EVEX.512.66.0F38.W0 14 /r VPRORVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F OR AVX10.1[1]	Rotate doublewords in zmm2 right by count in the corresponding element of zmm3/m512/m32bcst, store result using writemask k1.
EVEX.512.66.0F.W0 72 /0 ib VPRORD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	A	V/V	AVX512F OR AVX10.1	Rotate doublewords in zmm2/m512/m32bcst right by imm8, store result using writemask k1.
EVEX.512.66.0F38.W1 14 /r VPRORVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F OR AVX10.1	Rotate quadwords in zmm2 right by count in the corresponding element of zmm3/m512/m64bcst, store result using writemask k1.
EVEX.512.66.0F.W1 72 /0 ib VPRORQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	A	V/V	AVX512F OR AVX10.1	Rotate quadwords in zmm2/m512/m64bcst right by imm8, store result using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	VEX.vvvv (w)	ModRM:r/m (R)	imm8	N/A
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Rotates the bits in the individual data elements (doublewords, or quadword) in the first source operand to the right by the number of bits specified in the count operand. If the value specified by the count operand is greater than 31 (for doublewords), or 63 (for a quadword), then the count operand modulo the data size (32 or 64) is used.

EVEX.128 encoded version: The destination operand is a XMM register. The source operand is a XMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

EVEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The destination operand is a ZMM register updated according to the writemask. For the count operand in immediate form, the source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location, the count operand is an 8-bit immediate. For the count operand in variable form, the first source operand (the second operand) is a ZMM register and the counter operand (the third operand) is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location.

Operation

```
RIGHT_ROTATE_DWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC modulo 32;
DEST[31:0] := (SRC >> COUNT) | (SRC << (32 - COUNT));

RIGHT_ROTATE_QWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC modulo 64;
DEST[63:0] := (SRC >> COUNT) | (SRC << (64 - COUNT));
```

VPRORD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+31:i] := RIGHT_ROTATE_DWORDS( SRC1[31:0], imm8)
                ELSE DEST[i+31:i] := RIGHT_ROTATE_DWORDS(SRC1[i+31:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPRORVD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := RIGHT_ROTATE_DWORDS(SRC1[i+31:i], SRC2[31:0])
                ELSE DEST[i+31:i] := RIGHT_ROTATE_DWORDS(SRC1[i+31:i], SRC2[i+31:i])
            FI;
```

```
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPRORQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+63:i] := RIGHT_ROTATE_QWORDS(SRC1[63:0], imm8)
                ELSE DEST[i+63:i] := RIGHT_ROTATE_QWORDS(SRC1[i+63:i], imm8])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPRORVQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := RIGHT_ROTATE_QWORDS(SRC1[i+63:i], SRC2[63:0])
                ELSE DEST[i+63:i] := RIGHT_ROTATE_QWORDS(SRC1[i+63:i], SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPRORD __m512i _mm512_ror_epi32(__m512i a, int imm);
VPRORD __m512i _mm512_mask_ror_epi32(__m512i a, __mmask16 k, __m512i b, int imm);
VPRORD __m512i _mm512_maskz_ror_epi32(__mmask16 k, __m512i a, int imm);
VPRORD __m256i _mm256_ror_epi32(__m256i a, int imm);
VPRORD __m256i _mm256_mask_ror_epi32(__m256i a, __mmask8 k, __m256i b, int imm);
VPRORD __m256i _mm256_maskz_ror_epi32(__mmask8 k, __m256i a, int imm);
VPRORD __m128i _mm_ror_epi32(__m128i a, int imm);
VPRORD __m128i _mm_mask_ror_epi32(__m128i a, __mmask8 k, __m128i b, int imm);
VPRORD __m128i _mm_maskz_ror_epi32(__mmask8 k, __m128i a, int imm);
VPRORQ __m512i _mm512_ror_epi64(__m512i a, int imm);
VPRORQ __m512i _mm512_mask_ror_epi64(__m512i a, __mmask8 k, __m512i b, int imm);
VPRORQ __m512i _mm512_maskz_ror_epi64(__mmask8 k, __m512i a, int imm);
VPRORQ __m256i _mm256_ror_epi64(__m256i a, int imm);
VPRORQ __m256i _mm256_mask_ror_epi64(__m256i a, __mmask8 k, __m256i b, int imm);
VPRORQ __m256i _mm256_maskz_ror_epi64(__mmask8 k, __m256i a, int imm);
VPRORQ __m128i _mm_ror_epi64(__m128i a, int imm);
VPRORQ __m128i _mm_mask_ror_epi64(__m128i a, __mmask8 k, __m128i b, int imm);
VPRORQ __m128i _mm_maskz_ror_epi64(__mmask8 k, __m128i a, int imm);
VPRORVD __m512i _mm512_rorv_epi32(__m512i a, __m512i cnt);
VPRORVD __m512i _mm512_mask_rorv_epi32(__m512i a, __mmask16 k, __m512i b, __m512i cnt);
VPRORVD __m512i _mm512_maskz_rorv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPRORVD __m256i _mm256_rorv_epi32(__m256i a, __m256i cnt);
VPRORVD __m256i _mm256_mask_rorv_epi32(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPRORVD __m256i _mm256_maskz_rorv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPRORVD __m128i _mm_rorv_epi32(__m128i a, __m128i cnt);
VPRORVD __m128i _mm_mask_rorv_epi32(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPRORVD __m128i _mm_maskz_rorv_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPRORVQ __m512i _mm512_rorv_epi64(__m512i a, __m512i cnt);
VPRORVQ __m512i _mm512_mask_rorv_epi64(__m512i a, __mmask8 k, __m512i b, __m512i cnt);
VPRORVQ __m512i _mm512_maskz_rorv_epi64(__mmask8 k, __m512i a, __m512i cnt);
VPRORVQ __m256i _mm256_rorv_epi64(__m256i a, __m256i cnt);
VPRORVQ __m256i _mm256_mask_rorv_epi64(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPRORVQ __m256i _mm256_maskz_rorv_epi64(__mmask8 k, __m256i a, __m256i cnt);
VPRORVQ __m128i _mm_rorv_epi64(__m128i a, __m128i cnt);
VPRORVQ __m128i _mm_mask_rorv_epi64(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPRORVQ __m128i _mm_maskz_rorv_epi64(__mmask8 k, __m128i a, __m128i cnt);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

VPSCATTERDD/VPSCATTERDQ/VPSCATTERQD/VPSCATTERQQ—Scatter Packed Dword, Packed Qword with Signed Dword, Signed Qword Indices

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 A0 /vsib VPSCATTERDD vm32x {k1}, xmm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, scatter dword values to memory using writemask k1.
EVEX.256.66.0F38.W0 A0 /vsib VPSCATTERDD vm32y {k1}, ymm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, scatter dword values to memory using writemask k1.
EVEX.512.66.0F38.W0 A0 /vsib VPSCATTERDD vm32z {k1}, zmm1	A	V/V	AVX512F OR AVX10.1	Using signed dword indices, scatter dword values to memory using writemask k1.
EVEX.128.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32x {k1}, xmm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, scatter qword values to memory using writemask k1.
EVEX.256.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32x {k1}, ymm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, scatter qword values to memory using writemask k1.
EVEX.512.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32y {k1}, zmm1	A	V/V	AVX512F OR AVX10.1	Using signed dword indices, scatter qword values to memory using writemask k1.
EVEX.128.66.0F38.W0 A1 /vsib VPSCATTERQD vm64x {k1}, xmm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, scatter dword values to memory using writemask k1.
EVEX.256.66.0F38.W0 A1 /vsib VPSCATTERQD vm64y {k1}, xmm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, scatter dword values to memory using writemask k1.
EVEX.512.66.0F38.W0 A1 /vsib VPSCATTERQD vm64z {k1}, ymm1	A	V/V	AVX512F OR AVX10.1	Using signed qword indices, scatter dword values to memory using writemask k1.
EVEX.128.66.0F38.W1 A1 /vsib VPSCATTERQQ vm64x {k1}, xmm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, scatter qword values to memory using writemask k1.
EVEX.256.66.0F38.W1 A1 /vsib VPSCATTERQQ vm64y {k1}, ymm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, scatter qword values to memory using writemask k1.
EVEX.512.66.0F38.W1 A1 /vsib VPSCATTERQQ vm64z {k1}, zmm1	A	V/V	AVX512F OR AVX10.1	Using signed qword indices, scatter qword values to memory using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	ModRM:reg (r)	N/A	N/A

Description

Stores up to 16 elements (8 elements for qword indices) in doubleword vector or 8 elements in quadword vector to the memory locations pointed by base address BASE_ADDR and index vector VINDEX, with scale SCALE. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be stored if their corresponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already scattered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register are partially updated. If any traps or interrupts are pending from already scattered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

Note that:

- Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Note that this also include partially overlapping vector indices. Writes that are not overlapped may happen in any order. Memory ordering with other instructions follows the Intel-64 memory ordering model. Note that this does not account for non-overlapping indices that map into the same physical address locations.
- If two or more destination indices completely overlap, the "earlier" write(s) may be skipped.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination ZMM will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be scattered in any order, but faults must be delivered in a right-to left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- If this instruction overwrites itself and then takes a fault, only a subset of elements may be completed before the fault is delivered (as described above). If the fault handler completes and attempts to re-execute this instruction, the new instruction will be executed, and the scatter will not complete.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the k0 mask register is specified.

The instruction will #UD fault if EVEX.Z = 1.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1 or 4 byte displacement

VPSCATTERDD (EVEX encoded versions)

```
(KL, VL)= (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR +SignExtend(VINDEX[i+31:i]) * SCALE + DISP] := SRC[i+31:i]
            k1[j] := 0

    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
```

VPSCATTERDQ (EVEX encoded versions)

```
(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR +SignExtend(VINDEX[k+31:k]) * SCALE + DISP] := SRC[i+63:i]
            k1[j] := 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
```

VPSCATTERQD (EVEX encoded versions)

```
(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP] := SRC[i+31:i]
            k1[j] := 0

    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
```

VPSCATTERQQ (EVEX encoded versions)

```
(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + (VINDEX[j+63:j]) * SCALE + DISP] := SRC[i+63:i]
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSCATTERDD void _mm512_i32scatter_epi32(void * base, __m512i vdx, __m512i a, int scale);
VPSCATTERDD void _mm256_i32scatter_epi32(void * base, __m256i vdx, __m256i a, int scale);
VPSCATTERDD void _mm_i32scatter_epi32(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERDD void _mm512_mask_i32scatter_epi32(void * base, __mmask16 k, __m512i vdx, __m512i a, int scale);
VPSCATTERDD void _mm256_mask_i32scatter_epi32(void * base, __mmask8 k, __m256i vdx, __m256i a, int scale);
VPSCATTERDD void _mm_mask_i32scatter_epi32(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);
VPSCATTERDQ void _mm512_i32scatter_epi64(void * base, __m256i vdx, __m512i a, int scale);
VPSCATTERDQ void _mm256_i32scatter_epi64(void * base, __m128i vdx, __m256i a, int scale);
VPSCATTERDQ void _mm_i32scatter_epi64(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERDQ void _mm512_mask_i32scatter_epi64(void * base, __mmask8 k, __m256i vdx, __m512i a, int scale);
VPSCATTERDQ void _mm256_mask_i32scatter_epi64(void * base, __mmask8 k, __m128i vdx, __m256i a, int scale);
VPSCATTERDQ void _mm_mask_i32scatter_epi64(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);
VPSCATTERQD void _mm512_i64scatter_epi32(void * base, __m512i vdx, __m256i a, int scale);
VPSCATTERQD void _mm256_i64scatter_epi32(void * base, __m256i vdx, __m128i a, int scale);
VPSCATTERQD void _mm_i64scatter_epi32(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERQD void _mm512_mask_i64scatter_epi32(void * base, __mmask8 k, __m512i vdx, __m256i a, int scale);
VPSCATTERQD void _mm256_mask_i64scatter_epi32(void * base, __mmask8 k, __m256i vdx, __m128i a, int scale);
VPSCATTERQD void _mm_mask_i64scatter_epi32(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);

VPSCATTERQQ void _mm512_i64scatter_epi64(void * base, __m512i vdx, __m512i a, int scale);
VPSCATTERQQ void _mm256_i64scatter_epi64(void * base, __m256i vdx, __m256i a, int scale);
VPSCATTERQQ void _mm_i64scatter_epi64(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERQQ void _mm512_mask_i64scatter_epi64(void * base, __mmask8 k, __m512i vdx, __m512i a, int scale);
VPSCATTERQQ void _mm256_mask_i64scatter_epi64(void * base, __mmask8 k, __m256i vdx, __m256i a, int scale);
VPSCATTERQQ void _mm_mask_i64scatter_epi64(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, "Type E12 Class Exception Conditions."

VPSHLD—Concatenate and Shift Packed Data Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 70 /r /ib VPSHLDW xmm1{k1}{z}, xmm2, xmm3/m128, imm8	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into xmm1.
EVEX.256.66.0F3A.W1 70 /r /ib VPSHLDW ymm1{k1}{z}, ymm2, ymm3/m256, imm8	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into ymm1.
EVEX.512.66.0F3A.W1 70 /r /ib VPSHLDW zmm1{k1}{z}, zmm2, zmm3/m512, imm8	A	V/V	AVX512_VBMI2 OR AVX10.1	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into zmm1.
EVEX.128.66.0F3A.W0 71 /r /ib VPSHLDD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into xmm1.
EVEX.256.66.0F3A.W0 71 /r /ib VPSHLDD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into ymm1.
EVEX.512.66.0F3A.W0 71 /r /ib VPSHLDD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	B	V/V	AVX512_VBMI2 OR AVX10.1	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into zmm1.
EVEX.128.66.0F3A.W1 71 /r /ib VPSHLDQ xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into xmm1.
EVEX.256.66.0F3A.W1 71 /r /ib VPSHLDQ ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into ymm1.
EVEX.512.66.0F3A.W1 71 /r /ib VPSHLDQ zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	B	V/V	AVX512_VBMI2 OR AVX10.1	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

Concatenate packed data, extract result shifted to the left by constant value.

This instruction supports memory fault suppression.

Operation

VPSHLDW DEST, SRC2, SRC3, imm8

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        tmp := concat(SRC2.word[j], SRC3.word[j]) << (imm8 & 15)
        DEST.word[j] := tmp.word[1]
    ELSE IF *zeroing*:
        DEST.word[j] := 0
    *ELSE DEST.word[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VPSHLDD DEST, SRC2, SRC3, imm8

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 := SRC3.dword[0]
    ELSE:
        tsrc3 := SRC3.dword[j]
    IF MaskBit(j) OR *no writemask*:
        tmp := concat(SRC2.dword[j], tsrc3) << (imm8 & 31)
        DEST.dword[j] := tmp.dword[1]
    ELSE IF *zeroing*:
        DEST.dword[j] := 0
    *ELSE DEST.dword[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VPSHLDQ DEST, SRC2, SRC3, imm8

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 := SRC3.qword[0]
    ELSE:
        tsrc3 := SRC3.qword[j]
    IF MaskBit(j) OR *no writemask*:
        tmp := concat(SRC2.qword[j], tsrc3) << (imm8 & 63)
        DEST.qword[j] := tmp.qword[1]
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    *ELSE DEST.qword[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSHLDD __m128i _mm_shldi_epi32(__m128i, __m128i, int);
VPSHLDD __m128i _mm_mask_shldi_epi32(__m128i, __mmask8, __m128i, __m128i, int);
VPSHLDD __m128i _mm_maskz_shldi_epi32(__mmask8, __m128i, __m128i, int);
VPSHLDD __m256i _mm256_shldi_epi32(__m256i, __m256i, int);
VPSHLDD __m256i _mm256_mask_shldi_epi32(__m256i, __mmask8, __m256i, __m256i, int);
VPSHLDD __m256i _mm256_maskz_shldi_epi32(__mmask8, __m256i, __m256i, int);
VPSHLDD __m512i _mm512_shldi_epi32(__m512i, __m512i, int);
VPSHLDD __m512i _mm512_mask_shldi_epi32(__m512i, __mmask16, __m512i, __m512i, int);
VPSHLDD __m512i _mm512_maskz_shldi_epi32(__mmask16, __m512i, __m512i, int);
VPSHLDQ __m128i _mm_shldi_epi64(__m128i, __m128i, int);
VPSHLDQ __m128i _mm_mask_shldi_epi64(__m128i, __mmask8, __m128i, __m128i, int);
VPSHLDQ __m128i _mm_maskz_shldi_epi64(__mmask8, __m128i, __m128i, int);
VPSHLDQ __m256i _mm256_shldi_epi64(__m256i, __m256i, int);
VPSHLDQ __m256i _mm256_mask_shldi_epi64(__m256i, __mmask8, __m256i, __m256i, int);
VPSHLDQ __m256i _mm256_maskz_shldi_epi64(__mmask8, __m256i, __m256i, int);
VPSHLDQ __m512i _mm512_shldi_epi64(__m512i, __m512i, int);
VPSHLDQ __m512i _mm512_mask_shldi_epi64(__m512i, __mmask8, __m512i, __m512i, int);
VPSHLDQ __m512i _mm512_maskz_shldi_epi64(__mmask8, __m512i, __m512i, int);
VPSHLDW __m128i _mm_shldi_epi16(__m128i, __m128i, int);
VPSHLDW __m128i _mm_mask_shldi_epi16(__m128i, __mmask8, __m128i, __m128i, int);
VPSHLDW __m128i _mm_maskz_shldi_epi16(__mmask8, __m128i, __m128i, int);
VPSHLDW __m256i _mm256_shldi_epi16(__m256i, __m256i, int);
VPSHLDW __m256i _mm256_mask_shldi_epi16(__m256i, __mmask16, __m256i, __m256i, int);
VPSHLDW __m256i _mm256_maskz_shldi_epi16(__mmask16, __m256i, __m256i, int);
VPSHLDW __m512i _mm512_shldi_epi16(__m512i, __m512i, int);
VPSHLDW __m512i _mm512_mask_shldi_epi16(__m512i, __mmask32, __m512i, __m512i, int);
VPSHLDW __m512i _mm512_maskz_shldi_epi16(__mmask32, __m512i, __m512i, int);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, "Type E4 Class Exception Conditions."

VPSHLDV—Concatenate and Variable Shift Packed Data Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 70 /r VPSHLDVW xmm1{k1}{z}, xmm2, xmm3/m128	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate xmm1 and xmm2, extract result shifted to the left by value in xmm3/m128 into xmm1.
EVEX.256.66.0F38.W1 70 /r VPSHLDVW ymm1{k1}{z}, ymm2, ymm3/m256	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate ymm1 and ymm2, extract result shifted to the left by value in xmm3/m256 into ymm1.
EVEX.512.66.0F38.W1 70 /r VPSHLDVW zmm1{k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512_VBMI2 OR AVX10.1	Concatenate zmm1 and zmm2, extract result shifted to the left by value in zmm3/m512 into zmm1.
EVEX.128.66.0F38.W0 71 /r VPSHLDVD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate xmm1 and xmm2, extract result shifted to the left by value in xmm3/m128 into xmm1.
EVEX.256.66.0F38.W0 71 /r VPSHLDVD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate ymm1 and ymm2, extract result shifted to the left by value in xmm3/m256 into ymm1.
EVEX.512.66.0F38.W0 71 /r VPSHLDVD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512_VBMI2 OR AVX10.1	Concatenate zmm1 and zmm2, extract result shifted to the left by value in zmm3/m512 into zmm1.
EVEX.128.66.0F38.W1 71 /r VPSHLDVQ xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate xmm1 and xmm2, extract result shifted to the left by value in xmm3/m128 into xmm1.
EVEX.256.66.0F38.W1 71 /r VPSHLDVQ ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate ymm1 and ymm2, extract result shifted to the left by value in xmm3/m256 into ymm1.
EVEX.512.66.0F38.W1 71 /r VPSHLDVQ zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512_VBMI2 OR AVX10.1	Concatenate zmm1 and zmm2, extract result shifted to the left by value in zmm3/m512 into zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Concatenate packed data, extract result shifted to the left by variable value.

This instruction supports memory fault suppression.

Operation

```
FUNCTION concat(a,b):
    IF words:
        d.word[1] := a
        d.word[0] := b
        return d
    ELSE IF dwords:
        q.dword[1] := a
        q.dword[0] := b
        return q
    ELSE IF qwords:
        o.qword[1] := a
        o.qword[0] := b
        return o
```

VPSHLDVW DEST, SRC2, SRC3

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        tmp := concat(DEST.word[j], SRC2.word[j]) << (SRC3.word[j] & 15)
        DEST.word[j] := tmp.word[1]
    ELSE IF *zeroing*:
        DEST.word[j] := 0
    *ELSE DEST.word[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VPSHLDVD DEST, SRC2, SRC3

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 := SRC3.dword[0]
    ELSE:
        tsrc3 := SRC3.dword[j]
    IF MaskBit(j) OR *no writemask*:
        tmp := concat(DEST.dword[j], SRC2.dword[j]) << (tsrc3 & 31)
        DEST.dword[j] := tmp.dword[1]
    ELSE IF *zeroing*:
        DEST.dword[j] := 0
    *ELSE DEST.dword[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VPSHLDVQ DEST, SRC2, SRC3

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 := SRC3.qword[0]
    ELSE:
        tsrc3 := SRC3.qword[j]
    IF MaskBit(j) OR *no writemask*:
        tmp := concat(DEST.qword[j], SRC2.qword[j]) << (tsrc3 & 63)
        DEST.qword[j] := tmp.qword[1]
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    *ELSE DEST.qword[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSHLDVW __m128i _mm_shldv_epi16(__m128i, __m128i, __m128i);
VPSHLDVW __m128i _mm_mask_shldv_epi16(__m128i, __mmask8, __m128i, __m128i);
VPSHLDVW __m128i _mm_maskz_shldv_epi16(__mmask8, __m128i, __m128i, __m128i);
VPSHLDVW __m256i _mm256_shldv_epi16(__m256i, __m256i, __m256i);
VPSHLDVW __m256i _mm256_mask_shldv_epi16(__m256i, __mmask16, __m256i, __m256i);
VPSHLDVW __m256i _mm256_maskz_shldv_epi16(__mmask16, __m256i, __m256i, __m256i);
VPSHLDVQ __m512i _mm512_shldv_epi64(__m512i, __m512i, __m512i);
VPSHLDVQ __m512i _mm512_mask_shldv_epi64(__m512i, __mmask8, __m512i, __m512i);
VPSHLDVQ __m512i _mm512_maskz_shldv_epi64(__mmask8, __m512i, __m512i, __m512i);
VPSHLDVW __m128i _mm_shldv_epi16(__m128i, __m128i, __m128i);
VPSHLDVW __m128i _mm_mask_shldv_epi16(__m128i, __mmask8, __m128i, __m128i);
VPSHLDVW __m128i _mm_maskz_shldv_epi16(__mmask8, __m128i, __m128i, __m128i);
VPSHLDVW __m256i _mm256_shldv_epi16(__m256i, __m256i, __m256i);
VPSHLDVW __m256i _mm256_mask_shldv_epi16(__m256i, __mmask16, __m256i, __m256i);
VPSHLDVW __m256i _mm256_maskz_shldv_epi16(__mmask16, __m256i, __m256i, __m256i);
VPSHLDVW __m512i _mm512_shldv_epi16(__m512i, __m512i, __m512i);
VPSHLDVW __m512i _mm512_mask_shldv_epi16(__m512i, __mmask32, __m512i, __m512i);
VPSHLDVW __m512i _mm512_maskz_shldv_epi16(__mmask32, __m512i, __m512i, __m512i);
VPSHLDVD __m128i _mm_shldv_epi32(__m128i, __m128i, __m128i);
VPSHLDVD __m128i _mm_mask_shldv_epi32(__m128i, __mmask8, __m128i, __m128i);
VPSHLDVD __m128i _mm_maskz_shldv_epi32(__mmask8, __m128i, __m128i, __m128i);
VPSHLDVD __m256i _mm256_shldv_epi32(__m256i, __m256i, __m256i);
VPSHLDVD __m256i _mm256_mask_shldv_epi32(__m256i, __mmask8, __m256i, __m256i);
VPSHLDVD __m256i _mm256_maskz_shldv_epi32(__mmask8, __m256i, __m256i, __m256i);
VPSHLDVD __m512i _mm512_shldv_epi32(__m512i, __m512i, __m512i);
VPSHLDVD __m512i _mm512_mask_shldv_epi32(__m512i, __mmask16, __m512i, __m512i);
VPSHLDVD __m512i _mm512_maskz_shldv_epi32(__mmask16, __m512i, __m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, "Type E4 Class Exception Conditions."

VPSHRD—Concatenate and Shift Packed Data Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 72 /r /ib VPSHRDW xmm1{k1}{z}, xmm2, xmm3/m128, imm8	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into xmm1.
EVEX.256.66.0F3A.W1 72 /r /ib VPSHRDW ymm1{k1}{z}, ymm2, ymm3/m256, imm8	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into ymm1.
EVEX.512.66.0F3A.W1 72 /r /ib VPSHRDW zmm1{k1}{z}, zmm2, zmm3/m512, imm8	A	V/V	AVX512_VBMI2 OR AVX10.1	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into zmm1.
EVEX.128.66.0F3A.W0 73 /r /ib VPSHRDD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into xmm1.
EVEX.256.66.0F3A.W0 73 /r /ib VPSHRDD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into ymm1.
EVEX.512.66.0F3A.W0 73 /r /ib VPSHRDD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	B	V/V	AVX512_VBMI2 OR AVX10.1	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into zmm1.
EVEX.128.66.0F3A.W1 73 /r /ib VPSHRDQ xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into xmm1.
EVEX.256.66.0F3A.W1 73 /r /ib VPSHRDQ ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into ymm1.
EVEX.512.66.0F3A.W1 73 /r /ib VPSHRDQ zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	B	V/V	AVX512_VBMI2 OR AVX10.1	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

Concatenate packed data, extract result shifted to the right by constant value.

This instruction supports memory fault suppression.

Operation

VPSHRDW DEST, SRC2, SRC3, imm8

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        DEST.word[j] := concat(SRC3.word[j], SRC2.word[j]) >> (imm8 & 15)
    ELSE IF *zeroing*:
        DEST.word[j] := 0
    *ELSE DEST.word[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VPSHRDD DEST, SRC2, SRC3, imm8

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 := SRC3.dword[0]
    ELSE:
        tsrc3 := SRC3.dword[j]
    IF MaskBit(j) OR *no writemask*:
        DEST.dword[j] := concat(tsrc3, SRC2.dword[j]) >> (imm8 & 31)
    ELSE IF *zeroing*:
        DEST.dword[j] := 0
    *ELSE DEST.dword[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VPSHRDQ DEST, SRC2, SRC3, imm8

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 := SRC3.qword[0]
    ELSE:
        tsrc3 := SRC3.qword[j]
    IF MaskBit(j) OR *no writemask*:
        DEST.qword[j] := concat(tsrc3, SRC2.qword[j]) >> (imm8 & 63)
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    *ELSE DEST.qword[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSHRDQ __m128i _mm_shrdi_epi64(__m128i, __m128i, int);
VPSHRDQ __m128i _mm_mask_shrdi_epi64(__m128i, __mmask8, __m128i, __m128i, int);
VPSHRDQ __m128i _mm_maskz_shrdi_epi64(__mmask8, __m128i, __m128i, int);
VPSHRDQ __m256i _mm256_shrdi_epi64(__m256i, __m256i, int);
VPSHRDQ __m256i _mm256_mask_shrdi_epi64(__m256i, __mmask8, __m256i, __m256i, int);
VPSHRDQ __m256i _mm256_maskz_shrdi_epi64(__mmask8, __m256i, __m256i, int);
VPSHRDQ __m512i _mm512_shrdi_epi64(__m512i, __m512i, int);
VPSHRDQ __m512i _mm512_mask_shrdi_epi64(__m512i, __mmask8, __m512i, __m512i, int);
VPSHRDQ __m512i _mm512_maskz_shrdi_epi64(__mmask8, __m512i, __m512i, int);
VPSHRDD __m128i _mm_shrdi_epi32(__m128i, __m128i, int);
VPSHRDD __m128i _mm_mask_shrdi_epi32(__m128i, __mmask8, __m128i, __m128i, int);
VPSHRDD __m128i _mm_maskz_shrdi_epi32(__mmask8, __m128i, __m128i, int);
VPSHRDD __m256i _mm256_shrdi_epi32(__m256i, __m256i, int);
VPSHRDD __m256i _mm256_mask_shrdi_epi32(__m256i, __mmask8, __m256i, __m256i, int);
VPSHRDD __m256i _mm256_maskz_shrdi_epi32(__mmask8, __m256i, __m256i, int);
VPSHRDD __m512i _mm512_shrdi_epi32(__m512i, __m512i, int);
VPSHRDD __m512i _mm512_mask_shrdi_epi32(__m512i, __mmask16, __m512i, __m512i, int);
VPSHRDD __m512i _mm512_maskz_shrdi_epi32(__mmask16, __m512i, __m512i, int);
VPSHRDW __m128i _mm_shrdi_epi16(__m128i, __m128i, int);
VPSHRDW __m128i _mm_mask_shrdi_epi16(__m128i, __mmask8, __m128i, __m128i, int);
VPSHRDW __m128i _mm_maskz_shrdi_epi16(__mmask8, __m128i, __m128i, int);
VPSHRDW __m256i _mm256_shrdi_epi16(__m256i, __m256i, int);
VPSHRDW __m256i _mm256_mask_shrdi_epi16(__m256i, __mmask16, __m256i, __m256i, int);
VPSHRDW __m256i _mm256_maskz_shrdi_epi16(__mmask16, __m256i, __m256i, int);
VPSHRDW __m512i _mm512_shrdi_epi16(__m512i, __m512i, int);
VPSHRDW __m512i _mm512_mask_shrdi_epi16(__m512i, __mmask32, __m512i, __m512i, int);
VPSHRDW __m512i _mm512_maskz_shrdi_epi16(__mmask32, __m512i, __m512i, int);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, "Type E4 Class Exception Conditions."

VPSHRDV—Concatenate and Variable Shift Packed Data Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 72 /r VPSHRDVW xmm1{k1}{z}, xmm2, xmm3/m128	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate xmm1 and xmm2, extract result shifted to the right by value in xmm3/m128 into xmm1.
EVEX.256.66.0F38.W1 72 /r VPSHRDVW ymm1{k1}{z}, ymm2, ymm3/m256	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate ymm1 and ymm2, extract result shifted to the right by value in xmm3/m256 into ymm1.
EVEX.512.66.0F38.W1 72 /r VPSHRDVW zmm1{k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512_VBMI2 OR AVX10.1	Concatenate zmm1 and zmm2, extract result shifted to the right by value in zmm3/m512 into zmm1.
EVEX.128.66.0F38.W0 73 /r VPSHRDVD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate xmm1 and xmm2, extract result shifted to the right by value in xmm3/m128 into xmm1.
EVEX.256.66.0F38.W0 73 /r VPSHRDVD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate ymm1 and ymm2, extract result shifted to the right by value in xmm3/m256 into ymm1.
EVEX.512.66.0F38.W0 73 /r VPSHRDVD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512_VBMI2 OR AVX10.1	Concatenate zmm1 and zmm2, extract result shifted to the right by value in zmm3/m512 into zmm1.
EVEX.128.66.0F38.W1 73 /r VPSHRDVQ xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate xmm1 and xmm2, extract result shifted to the right by value in xmm3/m128 into xmm1.
EVEX.256.66.0F38.W1 73 /r VPSHRDVQ ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate ymm1 and ymm2, extract result shifted to the right by value in xmm3/m256 into ymm1.
EVEX.512.66.0F38.W1 73 /r VPSHRDVQ zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512_VBMI2 OR AVX10.1	Concatenate zmm1 and zmm2, extract result shifted to the right by value in zmm3/m512 into zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Concatenate packed data, extract result shifted to the right by variable value.

This instruction supports memory fault suppression.

Operation

VPSHRDVW DEST, SRC2, SRC3

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        DEST.word[j] := concat(SRC2.word[j], DEST.word[j]) >> (SRC3.word[j] & 15)
    ELSE IF *zeroing*:
        DEST.word[j] := 0
    *ELSE DEST.word[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VPSHRDVD DEST, SRC2, SRC3

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 := SRC3.dword[0]
    ELSE:
        tsrc3 := SRC3.dword[j]
    IF MaskBit(j) OR *no writemask*:
        DEST.dword[j] := concat(SRC2.dword[j], DEST.dword[j]) >> (tsrc3 & 31)
    ELSE IF *zeroing*:
        DEST.dword[j] := 0
    *ELSE DEST.dword[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VPSHRDVQ DEST, SRC2, SRC3

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 := SRC3.qword[0]
    ELSE:
        tsrc3 := SRC3.qword[j]
    IF MaskBit(j) OR *no writemask*:
        DEST.qword[j] := concat(SRC2.qword[j], DEST.qword[j]) >> (tsrc3 & 63)
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    *ELSE DEST.qword[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSHRDVQ __m128i _mm_shrdv_epi64(__m128i, __m128i, __m128i);
VPSHRDVQ __m128i _mm_mask_shrdv_epi64(__m128i, __mmask8, __m128i, __m128i);
VPSHRDVQ __m128i _mm_maskz_shrdv_epi64(__mmask8, __m128i, __m128i, __m128i);
VPSHRDVQ __m256i _mm256_shrdv_epi64(__m256i, __m256i, __m256i);
VPSHRDVQ __m256i _mm256_mask_shrdv_epi64(__m256i, __mmask8, __m256i, __m256i);
VPSHRDVQ __m256i _mm256_maskz_shrdv_epi64(__mmask8, __m256i, __m256i, __m256i);
VPSHRDVQ __m512i _mm512_shrdv_epi64(__m512i, __m512i, __m512i);
VPSHRDVQ __m512i _mm512_mask_shrdv_epi64(__m512i, __mmask8, __m512i, __m512i);
VPSHRDVQ __m512i _mm512_maskz_shrdv_epi64(__mmask8, __m512i, __m512i, __m512i);
VPSHRDVD __m128i _mm_shrdv_epi32(__m128i, __m128i, __m128i);
VPSHRDVD __m128i _mm_mask_shrdv_epi32(__m128i, __mmask8, __m128i, __m128i);
VPSHRDVD __m128i _mm_maskz_shrdv_epi32(__mmask8, __m128i, __m128i, __m128i);
VPSHRDVD __m256i _mm256_shrdv_epi32(__m256i, __m256i, __m256i);
VPSHRDVD __m256i _mm256_mask_shrdv_epi32(__m256i, __mmask8, __m256i, __m256i);
VPSHRDVD __m256i _mm256_maskz_shrdv_epi32(__mmask8, __m256i, __m256i, __m256i);
VPSHRDVD __m512i _mm512_shrdv_epi32(__m512i, __m512i, __m512i);
VPSHRDVD __m512i _mm512_mask_shrdv_epi32(__m512i, __mmask16, __m512i, __m512i);
VPSHRDVD __m512i _mm512_maskz_shrdv_epi32(__mmask16, __m512i, __m512i, __m512i);
VPSHRDVW __m128i _mm_shrdv_epi16(__m128i, __m128i, __m128i);
VPSHRDVW __m128i _mm_mask_shrdv_epi16(__m128i, __mmask8, __m128i, __m128i);
VPSHRDVW __m128i _mm_maskz_shrdv_epi16(__mmask8, __m128i, __m128i, __m128i);
VPSHRDVW __m256i _mm256_shrdv_epi16(__m256i, __m256i, __m256i);
VPSHRDVW __m256i _mm256_mask_shrdv_epi16(__m256i, __mmask16, __m256i, __m256i);
VPSHRDVW __m256i _mm256_maskz_shrdv_epi16(__mmask16, __m256i, __m256i, __m256i);
VPSHRDVW __m512i _mm512_shrdv_epi16(__m512i, __m512i, __m512i);
VPSHRDVW __m512i _mm512_mask_shrdv_epi16(__m512i, __mmask32, __m512i, __m512i);
VPSHRDVW __m512i _mm512_maskz_shrdv_epi16(__mmask32, __m512i, __m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, "Type E4 Class Exception Conditions."

VPSHUFBITQMB—Shuffle Bits From Quadword Elements Using Byte Indexes Into Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 8F /r VPSHUFBITQMB k1{k2}, xmm2, xmm3/m128	A	V/V	(AVX512_BITALG AND AVX512VL) OR AVX10.1	Extract values in xmm2 using control bits of xmm3/m128 with writemask k2 and load the result into mask register k1.
EVEX.256.66.0F38.W0 8F /r VPSHUFBITQMB k1{k2}, ymm2, ymm3/m256	A	V/V	(AVX512_BITALG AND AVX512VL) OR AVX10.1	Extract values in ymm2 using control bits of ymm3/m256 with writemask k2 and load the result into mask register k1.
EVEX.512.66.0F38.W0 8F /r VPSHUFBITQMB k1{k2}, zmm2, zmm3/m512	A	V/V	AVX512_BITALG OR AVX10.1	Extract values in zmm2 using control bits of zmm3/m512 with writemask k2 and load the result into mask register k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

The VPSHUFBITQMB instruction performs a bit gather select using second source as control and first source as data. The value of each output bit is identified using 6 control bits from the second source operand to select a data bit to be gathered from one of 64 contiguous bits of data in the first source operand. Each byte of the second source operand provides a 6-bit index that gives access to 64 bits (2^6=64) within each respective quadword 64-bit lane (i.e., index values in SRC2.qword[x] allow selecting bit values in SRC1.qword[x]).

Control data for each output bit is stored in 8 bit elements of SRC2, but only the 6 least significant bits of each element are used.

This instruction uses write masking according to the writemask k2 (zeroing only). This instruction supports memory fault suppression.

The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register or a memory location. The destination operand is mask register k1.

Operation

VPSHUFBITQMB DEST, SRC1, SRC2

```
(KL, VL) = (16,128), (32,256), (64, 512)
FOR i := 0 TO KL/8-1:                   //Qword
    FOR j := 0 to 7:                    // Byte
        IF k2[i*8+j] or *no writemask*:
            m := SRC2.qword[i].byte[j] & 0x3F
            k1[i*8+j] := SRC1.qword[i].bit[m]
        ELSE:
            k1[i*8+j] := 0
k1[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFBITQMB __mmask16 _mm_bitshuffle_epi64_mask(__m128i, __m128i);
VPSHUFBITQMB __mmask16 _mm_mask_bitshuffle_epi64_mask(__mmask16, __m128i, __m128i);
VPSHUFBITQMB __mmask32 _mm256_bitshuffle_epi64_mask(__m256i, __m256i);
VPSHUFBITQMB __mmask32 _mm256_mask_bitshuffle_epi64_mask(__mmask32, __m256i, __m256i);
VPSHUFBITQMB __mmask64 _mm512_bitshuffle_epi64_mask(__m512i, __m512i);
VPSHUFBITQMB __mmask64 _mm512_mask_bitshuffle_epi64_mask(__mmask64, __m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, "Type E4 Class Exception Conditions."

VPSLLVW/VPSLLVD/VPSLLVQ—Variable Bit Shift Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.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.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.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.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.
EVEX.128.66.0F38.W1 12 /r VPSLLVW xmm1 {k1}{z}, xmm2, xmm3/m128	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s using writemask k1.
EVEX.256.66.0F38.W1 12 /r VPSLLVW ymm1 {k1}{z}, ymm2, ymm3/m256	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s using writemask k1.
EVEX.512.66.0F38.W1 12 /r VPSLLVW zmm1 {k1}{z}, zmm2, zmm3/m512	B	V/V	AVX512BW OR AVX10.1	Shift words in zmm2 left by amount specified in the corresponding element of zmm3/m512 while shifting in 0s using writemask k1.
EVEX.128.66.0F38.W0 47 /r VPSLLVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in xmm2 left by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in 0s using writemask k1.
EVEX.256.66.0F38.W0 47 /r VPSLLVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in ymm2 left by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in 0s using writemask k1.
EVEX.512.66.0F38.W0 47 /r VPSLLVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Shift doublewords in zmm2 left by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in 0s using writemask k1.
EVEX.128.66.0F38.W1 47 /r VPSLLVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in xmm2 left by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in 0s using writemask k1.
EVEX.256.66.0F38.W1 47 /r VPSLLVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in ymm2 left by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in 0s using writemask k1.
EVEX.512.66.0F38.W1 47 /r VPSLLVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Shift quadwords in zmm2 left by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in 0s using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Shifts the bits in the individual data elements (words, doublewords or quadword) in the first source operand to the left by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSLLVD/Q: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX encoded VPSLLVW: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

Operation

VPSLLVW (EVEX encoded version)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := ZeroExtend(SRC1[i+15:i] << SRC2[i+15:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

VPSLLVD (VEX.128 version)

```
COUNT_0 := SRC2[31 : 0]
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 := SRC2[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[MAXVL-1:128] := 0;
```

VPSLLVD (VEX.256 version)

```
COUNT_0 := SRC2[31 : 0];
    (* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 := SRC2[255 : 224];
IF COUNT_0 < 32 THEN
DEST[31:0] := ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
DEST[31:0] := 0;
    (* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
DEST[255:224] := ZeroExtend(SRC1[255:224] << COUNT_7);
ELSE
DEST[255:224] := 0;
DEST[MAXVL-1:256] := 0;
```

VPSLLVD (EVEX encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := ZeroExtend(SRC1[i+31:i] << SRC2[31:0])
                ELSE DEST[i+31:i] := ZeroExtend(SRC1[i+31:i] << SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 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[MAXVL-1:128] := 0;
```

VPSLLVQ (VEX.256 version)

```
COUNT_0 := SRC2[63 : 0];
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 := SRC2[255 : 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;
DEST[MAXVL-1:256] := 0;
```

VPSLLVQ (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := ZeroExtend(SRC1[i+63:i] << SRC2[63:0])
                ELSE DEST[i+63:i] := ZeroExtend(SRC1[i+63:i] << SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSLLVW __m512i _mm512_sllv_epi16(__m512i a, __m512i cnt);
VPSLLVW __m512i _mm512_mask_sllv_epi16(__m512i s, __mmask32 k, __m512i a, __m512i cnt);
VPSLLVW __m512i _mm512_maskz_sllv_epi16(__mmask32 k, __m512i a, __m512i cnt);
VPSLLVW __m256i _mm256_mask_sllv_epi16(__m256i s, __mmask16 k, __m256i a, __m256i cnt);
VPSLLVW __m256i _mm256_maskz_sllv_epi16(__mmask16 k, __m256i a, __m256i cnt);
VPSLLVW __m128i _mm_mask_sllv_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLVW __m128i _mm_maskz_sllv_epi16(__mmask8 k, __m128i a, __m128i cnt);
VPSLLVD __m512i _mm512_sllv_epi32(__m512i a, __m512i cnt);
VPSLLVD __m512i _mm512_mask_sllv_epi32(__m512i s, __mmask16 k, __m512i a, __m512i cnt);
VPSLLVD __m512i _mm512_maskz_sllv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPSLLVD __m256i _mm256_mask_sllv_epi32(__m256i s, __mmask8 k, __m256i a, __m256i cnt);
VPSLLVD __m256i _mm256_maskz_sllv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPSLLVD __m128i _mm_mask_sllv_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLVD __m128i _mm_maskz_sllv_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPSLLVQ __m512i _mm512_sllv_epi64(__m512i a, __m512i cnt);
VPSLLVQ __m512i _mm512_mask_sllv_epi64(__m512i s, __mmask8 k, __m512i a, __m512i cnt);
VPSLLVQ __m512i _mm512_maskz_sllv_epi64(__mmask8 k, __m512i a, __m512i cnt);
VPSLLVD __m256i _mm256_mask_sllv_epi64(__m256i s, __mmask8 k, __m256i a, __m256i cnt);
VPSLLVD __m256i _mm256_maskz_sllv_epi64(__mmask8 k, __m256i a, __m256i cnt);
VPSLLVD __m128i _mm_mask_sllv_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLVD __m128i _mm_maskz_sllv_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSLLVD __m256i _mm256_sllv_epi32 (__m256i m, __m256i count)
VPSLLVQ __m256i _mm256_sllv_epi64 (__m256i m, __m256i count)

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded VPSLLVD/VPSLLVQ, see Table 2-51, “Type E4 Class Exception Conditions.”
EVEX-encoded VPSLLVW, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

VPSRAVW/VPSRAVD/VPSRAVQ—Variable Bit Shift Right Arithmetic

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.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 sign bits.
VEX.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 sign bits.
EVEX.128.66.0F38.W1 11 /r VPSRAVW xmm1 {k1}{z}, xmm2, xmm3/m128	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.256.66.0F38.W1 11 /r VPSRAVW ymm1 {k1}{z}, ymm2, ymm3/m256	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in sign bits using writemask k1.
EVEX.512.66.0F38.W1 11 /r VPSRAVW zmm1 {k1}{z}, zmm2, zmm3/m512	B	V/V	AVX512BW OR AVX10.1	Shift words in zmm2 right by amount specified in the corresponding element of zmm3/m512 while shifting in sign bits using writemask k1.
EVEX.128.66.0F38.W0 46 /r VPSRAVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in sign bits using writemask k1.
EVEX.256.66.0F38.W0 46 /r VPSRAVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in sign bits using writemask k1.
EVEX.512.66.0F38.W0 46 /r VPSRAVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Shift doublewords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in sign bits using writemask k1.
EVEX.128.66.0F38.W1 46 /r VPSRAVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in sign bits using writemask k1.
EVEX.256.66.0F38.W1 46 /r VPSRAVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in sign bits using writemask k1.
EVEX.512.66.0F38.W1 46 /r VPSRAVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Shift quadwords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in sign bits using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Shifts the bits in the individual data elements (word/doublewords/quadword) in the first source operand (the second operand) to the right by the number of bits specified in the count value of respective data elements in the second source operand (the third operand). As the bits in the data elements are shifted right, the empty high-order bits are set to the MSB (sign extension).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination data element is 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 (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX.512/256/128 encoded VPSRAVD/W: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX.512/256/128 encoded VPSRAVQ: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

Operation

VPSRAVW (EVEX encoded version)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN
            COUNT := SRC2[i+3:i]
            IF COUNT < 16
                THEN    DEST[i+15:i] := SignExtend(SRC1[i+15:i] >> COUNT)
                ELSE
                    FOR k := 0 TO 15
                        DEST[i+k] := SRC1[i+15]
                    ENDFOR;
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

VPSRAVD (VEX.128 version)

```
COUNT_0 := SRC2[31 : 0]
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 := SRC2[127 : 96];
DEST[31:0] := SignExtend(SRC1[31:0] >> COUNT_0);
```

```
    (* Repeat shift operation for 2nd through 4th dwords *)
DEST[127:96] := SignExtend(SRC1[127:96] >> COUNT_3);
DEST[MAXVL-1:128] := 0;
```

VPSRAVD (VEX.256 version)

```
COUNT_0 := SRC2[31 : 0];
    (* Repeat Each COUNT_i for the 2nd through 8th dwords of SRC2*)
COUNT_7 := SRC2[255 : 224];
DEST[31:0] := SignExtend(SRC1[31:0] >> COUNT_0);
    (* Repeat shift operation for 2nd through 7th dwords *)
DEST[255:224] := SignExtend(SRC1[255:224] >> COUNT_7);
DEST[MAXVL-1:256] := 0;
```

VPSRAVD (EVEX encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    COUNT := SRC2[4:0]
                    IF COUNT < 32
                        THEN    DEST[i+31:i] := SignExtend(SRC1[i+31:i] >> COUNT)
                        ELSE
                            FOR k := 0 TO 31
                                DEST[i+k] := SRC1[i+31]
                            ENDFOR;
                    FI
                ELSE
                    COUNT := SRC2[i+4:i]
                    IF COUNT < 32
                        THEN    DEST[i+31:i] := SignExtend(SRC1[i+31:i] >> COUNT)
                        ELSE
                            FOR k := 0 TO 31
                                DEST[i+k] := SRC1[i+31]
                            ENDFOR;
                    FI
            FI;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[31:0] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

VPSRAVQ (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
```

```
                THEN
                    COUNT := SRC2[5:0]
                    IF COUNT < 64
                        THEN    DEST[i+63:i] := SignExtend(SRC1[i+63:i] >> COUNT)
                        ELSE
                            FOR k := 0 TO 63
                                DEST[i+k] := SRC1[i+63]
                            ENDFOR;
                    FI
                ELSE
                    COUNT := SRC2[i+5:i]
                    IF COUNT < 64
                        THEN    DEST[i+63:i] := SignExtend(SRC1[i+63:i] >> COUNT)
                        ELSE
                            FOR k := 0 TO 63
                                DEST[i+k] := SRC1[i+63]
                            ENDFOR;
                    FI
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63:0] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSRAVD __m512i _mm512_srav_epi32(__m512i a, __m512i cnt);
VPSRAVD __m512i _mm512_mask_srav_epi32(__m512i s, __mmask16 m, __m512i a, __m512i cnt);
VPSRAVD __m512i _mm512_maskz_srav_epi32(__mmask16 m, __m512i a, __m512i cnt);
VPSRAVD __m256i _mm256_srav_epi32(__m256i a, __m256i cnt);
VPSRAVD __m256i _mm256_mask_srav_epi32(__m256i s, __mmask8 m, __m256i a, __m256i cnt);
VPSRAVD __m256i _mm256_maskz_srav_epi32(__mmask8 m, __m256i a, __m256i cnt);
VPSRAVD __m128i _mm_srav_epi32(__m128i a, __m128i cnt);
VPSRAVD __m128i _mm_mask_srav_epi32(__m128i s, __mmask8 m, __m128i a, __m128i cnt);
VPSRAVD __m128i _mm_maskz_srav_epi32(__mmask8 m, __m128i a, __m128i cnt);
VPSRAVQ __m512i _mm512_srav_epi64(__m512i a, __m512i cnt);
VPSRAVQ __m512i _mm512_mask_srav_epi64(__m512i s, __mmask8 m, __m512i a, __m512i cnt);
VPSRAVQ __m512i _mm512_maskz_srav_epi64(__mmask8 m, __m512i a, __m512i cnt);
VPSRAVQ __m256i _mm256_srav_epi64(__m256i a, __m256i cnt);
VPSRAVQ __m256i _mm256_mask_srav_epi64(__m256i s, __mmask8 m, __m256i a, __m256i cnt);
VPSRAVQ __m256i _mm256_maskz_srav_epi64(__mmask8 m, __m256i a, __m256i cnt);
VPSRAVQ __m128i _mm_srav_epi64(__m128i a, __m128i cnt);
VPSRAVQ __m128i _mm_mask_srav_epi64(__m128i s, __mmask8 m, __m128i a, __m128i cnt);
VPSRAVQ __m128i _mm_maskz_srav_epi64(__mmask8 m, __m128i a, __m128i cnt);
VPSRAVW __m512i _mm512_srav_epi16(__m512i a, __m512i cnt);
VPSRAVW __m512i _mm512_mask_srav_epi16(__m512i s, __mmask32 m, __m512i a, __m512i cnt);
VPSRAVW __m512i _mm512_maskz_srav_epi16(__mmask32 m, __m512i a, __m512i cnt);
VPSRAVW __m256i _mm256_srav_epi16(__m256i a, __m256i cnt);
VPSRAVW __m256i _mm256_mask_srav_epi16(__m256i s, __mmask16 m, __m256i a, __m256i cnt);
VPSRAVW __m256i _mm256_maskz_srav_epi16(__mmask16 m, __m256i a, __m256i cnt);
VPSRAVW __m128i _mm_srav_epi16(__m128i a, __m128i cnt);
VPSRAVW __m128i _mm_mask_srav_epi16(__m128i s, __mmask8 m, __m128i a, __m128i cnt);
VPSRAVW __m128i _mm_maskz_srav_epi32(__mmask8 m, __m128i a, __m128i cnt);
VPSRAVD __m256i _mm256_srav_epi32 (__m256i m, __m256i count)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-51, "Type E4 Class Exception Conditions."

VPSRLVW/VPSRLVD/VPSRLVQ—Variable Bit Shift Right Logical

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.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.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.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.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.
EVEX.128.66.0F38.W1 10 /r VPSRLVW xmm1 {k1}{z}, xmm2, xmm3/m128	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s using writemask k1.
EVEX.256.66.0F38.W1 10 /r VPSRLVW ymm1 {k1}{z}, ymm2, ymm3/m256	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s using writemask k1.
EVEX.512.66.0F38.W1 10 /r VPSRLVW zmm1 {k1}{z}, zmm2, zmm3/m512	B	V/V	AVX512BW OR AVX10.1	Shift words in zmm2 right by amount specified in the corresponding element of zmm3/m512 while shifting in 0s using writemask k1.
EVEX.128.66.0F38.W0 45 /r VPSRLVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in 0s using writemask k1.
EVEX.256.66.0F38.W0 45 /r VPSRLVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in 0s using writemask k1.
EVEX.512.66.0F38.W0 45 /r VPSRLVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Shift doublewords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in 0s using writemask k1.
EVEX.128.66.0F38.W1 45 /r VPSRLVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in 0s using writemask k1.
EVEX.256.66.0F38.W1 45 /r VPSRLVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in 0s using writemask k1.
EVEX.512.66.0F38.W1 45 /r VPSRLVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Shift quadwords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in 0s using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Shifts the bits in the individual data elements (words, doublewords or quadword) in the first source operand to the right by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSRLVD/Q: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX encoded VPSRLVW: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

Operation

VPSRLVW (EVEX encoded version)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := ZeroExtend(SRC1[i+15:i] >> SRC2[i+15:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

VPSRLVD (VEX.128 version)

```
COUNT_0 := SRC2[31 : 0]
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 := SRC2[127 : 96];
IF COUNT_0 < 32 THEN
    DEST[31:0] := ZeroExtend(SRC1[31:0] >> COUNT_0);
ELSE
    DEST[31:0] := 0;
    (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 32 THEN
    DEST[127:96] := ZeroExtend(SRC1[127:96] >> COUNT_3);
ELSE
    DEST[127:96] := 0;
DEST[MAXVL-1:128] := 0;
```

VPSRLVD (VEX.256 version)

```
COUNT_0 := SRC2[31 : 0];
    (* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 := SRC2[255 : 224];
IF COUNT_0 < 32 THEN
DEST[31:0] := ZeroExtend(SRC1[31:0] >> COUNT_0);
ELSE
DEST[31:0] := 0;
    (* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
    DEST[255:224] := ZeroExtend(SRC1[255:224] >> COUNT_7);
ELSE
    DEST[255:224] := 0;
DEST[MAXVL-1:256] := 0;
```

VPSRLVD (EVEX encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := ZeroExtend(SRC1[i+31:i] >> SRC2[31:0])
                ELSE DEST[i+31:i] := ZeroExtend(SRC1[i+31:i] >> SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

VPSRLVQ (VEX.128 version)

```
COUNT_0 := SRC2[63 : 0];
COUNT_1 := SRC2[127 : 64];
IF COUNT_0 < 64 THEN
    DEST[63:0] := ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
    DEST[63:0] := 0;
IF COUNT_1 < 64 THEN
    DEST[127:64] := ZeroExtend(SRC1[127:64] >> COUNT_1);
ELSE
    DEST[127:64] := 0;
DEST[MAXVL-1:128] := 0;
```

VPSRLVQ (VEX.256 version)

```
COUNT_0 := SRC2[63 : 0];
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 := SRC2[255 : 192];
IF COUNT_0 < 64 THEN
DEST[63:0] := ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
DEST[63:0] := 0;
    (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64 THEN
    DEST[255:192] := ZeroExtend(SRC1[255:192] >> COUNT_3);
ELSE
    DEST[255:192] := 0;
DEST[MAXVL-1:256] := 0;
```

VPSRLVQ (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := ZeroExtend(SRC1[i+63:i] >> SRC2[63:0])
                ELSE DEST[i+63:i] := ZeroExtend(SRC1[i+63:i] >> SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSRLVW __m512i _mm512_srlv_epi16(__m512i a, __m512i cnt);
VPSRLVW __m512i _mm512_mask_srlv_epi16(__m512i s, __mmask32 k, __m512i a, __m512i cnt);
VPSRLVW __m512i _mm512_maskz_srlv_epi16(__mmask32 k, __m512i a, __m512i cnt);
VPSRLVW __m256i _mm256_mask_srlv_epi16(__m256i s, __mmask16 k, __m256i a, __m256i cnt);
VPSRLVW __m256i _mm256_maskz_srlv_epi16(__mmask16 k, __m256i a, __m256i cnt);
VPSRLVW __m128i _mm_mask_srlv_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLVW __m128i _mm_maskz_srlv_epi16(__mmask8 k, __m128i a, __m128i cnt);
VPSRLVW __m256i _mm256_srlv_epi32 (__m256i m, __m256i count)
VPSRLVD __m512i _mm512_srlv_epi32(__m512i a, __m512i cnt);
VPSRLVD __m512i _mm512_mask_srlv_epi32(__m512i s, __mmask16 k, __m512i a, __m512i cnt);
VPSRLVD __m512i _mm512_maskz_srlv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPSRLVD __m256i _mm256_mask_srlv_epi32(__m256i s, __mmask8 k, __m256i a, __m256i cnt);
VPSRLVD __m256i _mm256_maskz_srlv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPSRLVD __m128i _mm_mask_srlv_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLVD __m128i _mm_maskz_srlv_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPSRLVQ __m512i _mm512_srlv_epi64(__m512i a, __m512i cnt);
VPSRLVQ __m512i _mm512_mask_srlv_epi64(__m512i s, __mmask8 k, __m512i a, __m512i cnt);
VPSRLVQ __m512i _mm512_maskz_srlv_epi64(__mmask8 k, __m512i a, __m512i cnt);
VPSRLVQ __m256i _mm256_mask_srlv_epi64(__m256i s, __mmask8 k, __m256i a, __m256i cnt);
VPSRLVQ __m256i _mm256_maskz_srlv_epi64(__mmask8 k, __m256i a, __m256i cnt);
VPSRLVQ __m128i _mm_mask_srlv_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLVQ __m128i _mm_maskz_srlv_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSRLVQ __m256i _mm256_srlv_epi64 (__m256i m, __m256i count)
VPSRLVD __m128i _mm_srlv_epi32(__m128i a, __m128i cnt);
VPSRLVQ __m128i _mm_srlv_epi64(__m128i a, __m128i cnt);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instructions, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded VPSRLVD/Q, see Table 2-51, "Type E4 Class Exception Conditions."
EVEX-encoded VPSRLVW, see Exceptions Type E4.nb in Table 2-51, "Type E4 Class Exception Conditions."

VPTERNLOGD/VPTERNLOGQ—Bitwise Ternary Logic

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 25 /r ib VPTERNLOGD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise ternary logic taking xmm1, xmm2, and xmm3/m128/m32bcst as source operands and writing the result to xmm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.
EVEX.256.66.0F3A.W0 25 /r ib VPTERNLOGD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise ternary logic taking ymm1, ymm2, and ymm3/m256/m32bcst as source operands and writing the result to ymm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.
EVEX.512.66.0F3A.W0 25 /r ib VPTERNLOGD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	A	V/V	AVX512F OR AVX10.1	Bitwise ternary logic taking zmm1, zmm2, and zmm3/m512/m32bcst as source operands and writing the result to zmm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.
EVEX.128.66.0F3A.W1 25 /r ib VPTERNLOGQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise ternary logic taking xmm1, xmm2, and xmm3/m128/m64bcst as source operands and writing the result to xmm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.
EVEX.256.66.0F3A.W1 25 /r ib VPTERNLOGQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise ternary logic taking ymm1, ymm2, and ymm3/m256/m64bcst as source operands and writing the result to ymm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.
EVEX.512.66.0F3A.W1 25 /r ib VPTERNLOGQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	A	V/V	AVX512F OR AVX10.1	Bitwise ternary logic taking zmm1, zmm2, and zmm3/m512/m64bcst as source operands and writing the result to zmm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

VPTERNLOGD/Q takes three bit vectors of 512-bit length (in the first, second, and third operand) as input data to form a set of 512 indices, each index is comprised of one bit from each input vector. The imm8 byte specifies a boolean logic table producing a binary value for each 3-bit index value. The final 512-bit boolean result is written to the destination operand (the first operand) using the writemask k1 with the granularity of doubleword element or quadword element into the destination.

The destination operand is a ZMM (EVEX.512)/YMM (EVEX.256)/XMM (EVEX.128) register. The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location The destination operand is a ZMM register conditionally updated with writemask k1.

Table 5-20 shows two examples of Boolean functions specified by immediate values 0xE2 and 0xE4, with the look up result listed in the fourth column following the three columns containing all possible values of the 3-bit index.

Table 5-20. Examples of VPTERNLOGD/Q Imm8 Boolean Function and Input Index Values

VPTERNLOGD reg1, reg2, src3, 0xE2			Bit Result with Imm8=0xE2	VPTERNLOGD reg1, reg2, src3, 0xE4			Bit Result with Imm8=0xE4
Bit(reg1)	Bit(reg2)	Bit(src3)		Bit(reg1)	Bit(reg2)	Bit(src3)	
0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	1
0	1	1	0	0	1	1	0
1	0	0	0	1	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

Specifying different values in imm8 will allow any arbitrary three-input Boolean functions to be implemented in software using VPTERNLOGD/Q. Table 5-1 and Table 5-2 provide a mapping of all 256 possible imm8 values to various Boolean expressions.

Operation

VPTERNLOGD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            FOR k := 0 TO 31
                IF (EVEX.b = 1) AND (SRC2 *is memory*)
                    THEN DEST[j][k] := imm[(DEST[i+k] << 2) + (SRC1[ i+k ] << 1) + SRC2[ k ]]
                    ELSE DEST[j][k] := imm[(DEST[i+k] << 2) + (SRC1[ i+k ] << 1) + SRC2[ i+k ]]
                FI;
                        ; table lookup of immediate bellow;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31+i:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31+i:i] := 0
        FI;
    FI;
ENDFOR;
```

```
DEST[MAXVL-1:VL] := 0
```

VPTERNLOGQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            FOR k := 0 TO 63
                IF (EVEX.b = 1) AND (SRC2 *is memory*)
                    THEN DEST[j][k] := imm[(DEST[i+k] << 2) + (SRC1[ i+k ] << 1) + SRC2[ k ]]
                    ELSE DEST[j][k] := imm[(DEST[i+k] << 2) + (SRC1[ i+k ] << 1) + SRC2[ i+k ]]
                FI;                     ; table lookup of immediate bellow;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[63+i:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[63+i:i] := 0
            FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPTERNLOGD __m512i _mm512_ternarylogic_epi32(__m512i a, __m512i b, int imm);
VPTERNLOGD __m512i _mm512_mask_ternarylogic_epi32(__m512i s, __mmask16 m, __m512i a, __m512i b, int imm);
VPTERNLOGD __m512i _mm512_maskz_ternarylogic_epi32(__mmask m, __m512i a, __m512i b, int imm);
VPTERNLOGD __m256i _mm256_ternarylogic_epi32(__m256i a, __m256i b, int imm);
VPTERNLOGD __m256i _mm256_mask_ternarylogic_epi32(__m256i s, __mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGD __m256i _mm256_maskz_ternarylogic_epi32( __mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGD __m128i _mm_ternarylogic_epi32(__m128i a, __m128i b, int imm);
VPTERNLOGD __m128i _mm_mask_ternarylogic_epi32(__m128i s, __mmask8 m, __m128i a, __m128i b, int imm);
VPTERNLOGD __m128i _mm_maskz_ternarylogic_epi32( __mmask8 m, __m128i a, __m128i b, int imm);
VPTERNLOGQ __m512i _mm512_ternarylogic_epi64(__m512i a, __m512i b, int imm);
VPTERNLOGQ __m512i _mm512_mask_ternarylogic_epi64(__m512i s, __mmask8 m, __m512i a, __m512i b, int imm);
VPTERNLOGQ __m512i _mm512_maskz_ternarylogic_epi64( __mmask8 m, __m512i a, __m512i b, int imm);
VPTERNLOGQ __m256i _mm256_ternarylogic_epi64(__m256i a, __m256i b, int imm);
VPTERNLOGQ __m256i _mm256_mask_ternarylogic_epi64(__m256i s, __mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGQ __m256i _mm256_maskz_ternarylogic_epi64( __mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGQ __m128i _mm_ternarylogic_epi64(__m128i a, __m128i b, int imm);
VPTERNLOGQ __m128i _mm_mask_ternarylogic_epi64(__m128i s, __mmask8 m, __m128i a, __m128i b, int imm);
VPTERNLOGQ __m128i _mm_maskz_ternarylogic_epi64( __mmask8 m, __m128i a, __m128i b, int imm);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, "Type E4 Class Exception Conditions."

VPTESTMB/VPTESTMW/VPTESTMD/VPTESTMQ—Logical AND and Set Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 26 /r VPTESTMB k2 {k1}, xmm2, xmm3/m128	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Bitwise AND of packed byte integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.256.66.0F38.W0 26 /r VPTESTMB k2 {k1}, ymm2, ymm3/m256	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Bitwise AND of packed byte integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.512.66.0F38.W0 26 /r VPTESTMB k2 {k1}, zmm2, zmm3/m512	A	V/V	AVX512BW OR AVX10.1	Bitwise AND of packed byte integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.128.66.0F38.W1 26 /r VPTESTMW k2 {k1}, xmm2, xmm3/m128	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Bitwise AND of packed word integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.256.66.0F38.W1 26 /r VPTESTMW k2 {k1}, ymm2, ymm3/m256	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Bitwise AND of packed word integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.512.66.0F38.W1 26 /r VPTESTMW k2 {k1}, zmm2, zmm3/m512	A	V/V	AVX512BW OR AVX10.1	Bitwise AND of packed word integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.128.66.0F38.W0 27 /r VPTESTMD k2 {k1}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise AND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.256.66.0F38.W0 27 /r VPTESTMD k2 {k1}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise AND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.512.66.0F38.W0 27 /r VPTESTMD k2 {k1}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F OR AVX10.1	Bitwise AND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.128.66.0F38.W1 27 /r VPTESTMQ k2 {k1}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise AND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.256.66.0F38.W1 27 /r VPTESTMQ k2 {k1}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise AND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.512.66.0F38.W1 27 /r VPTESTMQ k2 {k1}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F OR AVX10.1	Bitwise AND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical AND operation on the first source operand (the second operand) and second source operand (the third operand) and stores the result in the destination operand (the first operand) under the write-mask. Each bit of the result is set to 1 if the bitwise AND of the corresponding elements of the first and second src operands is non-zero; otherwise it is set to 0.

VPTESTMD/VPTESTMQ: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a mask register updated under the writemask.

VPTESTMB/VPTESTMW: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a mask register updated under the writemask.

Operation

VPTESTMB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN    DEST[j] := (SRC1[i+7:i] BITWISE AND SRC2[i+7:i] != 0)? 1 : 0;
        ELSE    DEST[j] = 0                     ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPTESTMW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN    DEST[j] := (SRC1[i+15:i] BITWISE AND SRC2[i+15:i] != 0)? 1 : 0;
        ELSE    DEST[j] = 0                     ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPTESTMD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[j] := (SRC1[i+31:i] BITWISE AND SRC2[31:0] != 0)? 1 : 0;
                ELSE DEST[j] := (SRC1[i+31:i] BITWISE AND SRC2[i+31:i] != 0)? 1 : 0;
            FI;
        ELSE    DEST[j] := 0                    ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPTESTMQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[j] := (SRC1[i+63:i] BITWISE AND SRC2[63:0] != 0)? 1 : 0;
                ELSE DEST[j] := (SRC1[i+63:i] BITWISE AND SRC2[i+63:i] != 0)? 1 : 0;
            FI;
        ELSE    DEST[j] := 0                    ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPTESTMB __mmask64 _mm512_test_epi8_mask(__m512i a, __m512i b);
VPTESTMB __mmask64 _mm512_mask_test_epi8_mask(__mmask64, __m512i a, __m512i b);
VPTESTMW __mmask32 _mm512_test_epi16_mask(__m512i a, __m512i b);
VPTESTMW __mmask32 _mm512_mask_test_epi16_mask(__mmask32, __m512i a, __m512i b);
VPTESTMD __mmask16 _mm512_test_epi32_mask(__m512i a, __m512i b);
VPTESTMD __mmask16 _mm512_mask_test_epi32_mask(__mmask16, __m512i a, __m512i b);
VPTESTMQ __mmask8 _mm512_test_epi64_mask(__m512i a, __m512i b);
VPTESTMQ __mmask8 _mm512_mask_test_epi64_mask(__mmask8, __m512i a, __m512i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VPTESTMD/Q: See Table 2-51, "Type E4 Class Exception Conditions."
VPTESTMB/W: See Exceptions Type E4.nb in Table 2-51, "Type E4 Class Exception Conditions."

VPTESTNMB/W/D/Q—Logical NAND and Set

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 26 /r VPTESTNMB k2 {k1}, xmm2, xmm3/m128	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Bitwise NAND of packed byte integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.256.F3.0F38.W0 26 /r VPTESTNMB k2 {k1}, ymm2, ymm3/m256	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Bitwise NAND of packed byte integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.512.F3.0F38.W0 26 /r VPTESTNMB k2 {k1}, zmm2, zmm3/m512	A	V/V	(AVX512F AND AVX512BW) OR AVX10.1	Bitwise NAND of packed byte integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.128.F3.0F38.W1 26 /r VPTESTNMW k2 {k1}, xmm2, xmm3/m128	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Bitwise NAND of packed word integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.256.F3.0F38.W1 26 /r VPTESTNMW k2 {k1}, ymm2, ymm3/m256	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Bitwise NAND of packed word integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.512.F3.0F38.W1 26 /r VPTESTNMW k2 {k1}, zmm2, zmm3/m512	A	V/V	(AVX512F AND AVX512BW) OR AVX10.1	Bitwise NAND of packed word integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.128.F3.0F38.W0 27 /r VPTESTNMD k2 {k1}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise NAND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.256.F3.0F38.W0 27 /r VPTESTNMD k2 {k1}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise NAND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.512.F3.0F38.W0 27 /r VPTESTNMD k2 {k1}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F OR AVX10.1	Bitwise NAND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.128.F3.0F38.W1 27 /r VPTESTNMQ k2 {k1}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise NAND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.256.F3.0F38.W1 27 /r VPTESTNMQ k2 {k1}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise NAND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.512.F3.0F38.W1 27 /r VPTESTNMQ k2 {k1}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F OR AVX10.1	Bitwise NAND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical NAND operation on the byte/word/doubleword/quadword element of the first source operand (the second operand) with the corresponding element of the second source operand (the third operand) and stores the logical comparison result into each bit of the destination operand (the first operand) according to the writemask k1. Each bit of the result is set to 1 if the bitwise AND of the corresponding elements of the first and second src operands is zero; otherwise it is set to 0.

EVEX encoded VPTESTNMD/Q: The first source operand is a ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination is updated according to the writemask.

EVEX encoded VPTESTNMB/W: The first source operand is a ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.

Operation

VPTESTNMB

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j*8
    IF MaskBit(j) OR *no writemask*
        THEN
            DEST[j] := (SRC1[i+7:i] BITWISE AND SRC2[i+7:i] == 0)? 1 : 0
        ELSE DEST[j] := 0; zeroing masking only
    FI
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPTESTNMW

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j*16
    IF MaskBit(j) OR *no writemask*
        THEN
            DEST[j] := (SRC1[i+15:i] BITWISE AND SRC2[i+15:i] == 0)? 1 : 0
        ELSE DEST[j] := 0; zeroing masking only
    FI
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPTESTNMD

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j*32
    IF MaskBit(j) OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := (SRC1[i+31:i] BITWISE AND SRC2[31:0] == 0)? 1 : 0
                ELSE DEST[j] := (SRC1[i+31:i] BITWISE AND SRC2[i+31:i] == 0)? 1 : 0
            FI
        ELSE DEST[j] := 0; zeroing masking only
    FI
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPTESTNMQ

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j*64
    IF MaskBit(j) OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[j] := (SRC1[i+63:i] BITWISE AND SRC2[63:0] == 0)? 1 : 0;
                ELSE DEST[j] := (SRC1[i+63:i] BITWISE AND SRC2[i+63:i] == 0)? 1 : 0;
            FI;
        ELSE DEST[j] := 0; zeroing masking only
    FI
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPTESTNMB __mmask64 _mm512_testn_epi8_mask(__m512i a, __m512i b);
VPTESTNMB __mmask64 _mm512_mask_testn_epi8_mask(__mmask64, __m512i a, __m512i b);
VPTESTNMB __mmask32 _mm256_testn_epi8_mask(__m256i a, __m256i b);
VPTESTNMB __mmask32 _mm256_mask_testn_epi8_mask(__mmask32, __m256i a, __m256i b);
VPTESTNMB __mmask16 _mm_testn_epi8_mask(__m128i a, __m128i b);
VPTESTNMB __mmask16 _mm_mask_testn_epi8_mask(__mmask16, __m128i a, __m128i b);
VPTESTNMW __mmask32 _mm512_testn_epi16_mask(__m512i a, __m512i b);
VPTESTNMW __mmask32 _mm512_mask_testn_epi16_mask(__mmask32, __m512i a, __m512i b);
VPTESTNMW __mmask16 _mm256_testn_epi16_mask(__m256i a, __m256i b);
VPTESTNMW __mmask16 _mm256_mask_testn_epi16_mask(__mmask16, __m256i a, __m256i b);
VPTESTNMW __mmask8 _mm_testn_epi16_mask(__m128i a, __m128i b);
VPTESTNMW __mmask8 _mm_mask_testn_epi16_mask(__mmask8, __m128i a, __m128i b);
VPTESTNMD __mmask16 _mm512_testn_epi32_mask(__m512i a, __m512i b);
VPTESTNMD __mmask16 _mm512_mask_testn_epi32_mask(__mmask16, __m512i a, __m512i b);
VPTESTNMD __mmask8 _mm256_testn_epi32_mask(__m256i a, __m256i b);
VPTESTNMD __mmask8 _mm256_mask_testn_epi32_mask(__mmask8, __m256i a, __m256i b);
VPTESTNMD __mmask8 _mm_testn_epi32_mask(__m128i a, __m128i b);
VPTESTNMD __mmask8 _mm_mask_testn_epi32_mask(__mmask8, __m128i a, __m128i b);
VPTESTNMQ __mmask8 _mm512_testn_epi64_mask(__m512i a, __m512i b);
VPTESTNMQ __mmask8 _mm512_mask_testn_epi64_mask(__mmask8, __m512i a, __m512i b);
VPTESTNMQ __mmask8 _mm256_testn_epi64_mask(__m256i a, __m256i b);
VPTESTNMQ __mmask8 _mm256_mask_testn_epi64_mask(__mmask8, __m256i a, __m256i b);
VPTESTNMQ __mmask8 _mm_testn_epi64_mask(__m128i a, __m128i b);

VPTESTNMQ __mmask8 _mm_mask_testn_epi64_mask(__mmask8, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VPTESTNMD/VPTESTNMQ: See Table 2-51, "Type E4 Class Exception Conditions."

VPTESTNMB/VPTESTNMW: See Exceptions Type E4.nb in Table 2-51, "Type E4 Class Exception Conditions."

VRANGEPD—Range Restriction Calculation for Packed Pairs of Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 50 /r ib VRANGEPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Calculate two RANGE operation output value from 2 pairs of double precision floating-point values in xmm2 and xmm3/m128/m32bcst, store the results to xmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.
EVEX.256.66.0F3A.W1 50 /r ib VRANGEPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Calculate four RANGE operation output value from 4pairs of double precision floating-point values in ymm2 and ymm3/m256/m32bcst, store the results to ymm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.
EVEX.512.66.0F3A.W1 50 /r ib VRANGEPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}, imm8	A	V/V	AVX512DQ OR AVX10.1	Calculate eight RANGE operation output value from 8 pairs of double precision floating-point values in zmm2 and zmm3/m512/m32bcst, store the results to zmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

This instruction calculates 2/4/8 range operation outputs from two sets of packed input double precision floating-point values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of imm8[1:0] and imm8[3:2] are shown in Figure 5-27.

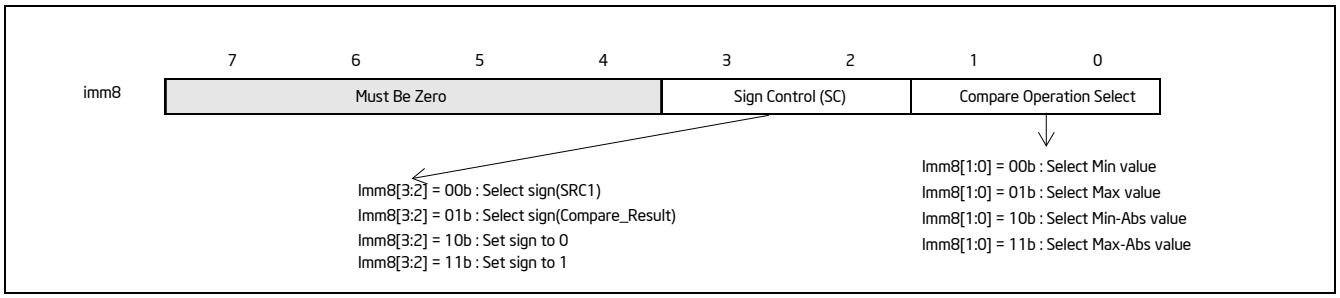


Figure 5-27. Imm8 Controls for VRANGEPD/SD/PS/SS

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-21. If the comparison raises an IE, the sign select control (imm8[3:2]) has no effect to the range operation output; this is indicated also in Table 5-21.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar floating-point MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-22.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-23.

Table 5-21. Signaling of Comparison Operation of One or More NaN Input Values and Effect of Imm8[3:2]

Src1	Src2	Result	IE Signaling Due to Comparison	Imm8[3:2] Effect to Range Output
sNaN1	sNaN2	Quiet(sNaN1)	Yes	Ignored
sNaN1	qNaN2	Quiet(sNaN1)	Yes	Ignored
sNaN1	Norm2	Quiet(sNaN1)	Yes	Ignored
qNaN1	sNaN2	Quiet(sNaN2)	Yes	Ignored
qNaN1	qNaN2	qNaN1	No	Applicable
qNaN1	Norm2	Norm2	No	Applicable
Norm1	sNaN2	Quiet(sNaN2)	Yes	Ignored
Norm1	qNaN2	Norm1	No	Applicable

Table 5-22. Comparison Result for Opposite-Signed Zero Cases for MIN, MIN_ABS, and MAX, MAX_ABS

MIN and MIN_ABS			MAX and MAX_ABS		
Src1	Src2	Result	Src1	Src2	Result
+0	-0	-0	+0	-0	+0
-0	+0	-0	-0	+0	+0

Table 5-23. Comparison Result of Equal-Magnitude Input Cases for MIN_ABS and MAX_ABS, (|a| = |b|, a>0, b<0)

MIN_ABS (|a| = |b|, a>0, b<0)			MAX_ABS (|a| = |b|, a>0, b<0)		
Src1	Src2	Result	Src1	Src2	Result
a	b	b	a	b	a
b	a	b	b	a	a

Operation

```
RangeDP(SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-21
    IF (SRC1 = SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2 = SNAN) THEN RETURN (QNAN(SRC2), set IE);

    Src1.exp := SRC1[62:52];
    Src1.fraction := SRC1[51:0];
    IF ((Src1.exp = 0 ) and (Src1.fraction != 0)) THEN// Src1 is a denormal number
        IF DAZ THEN Src1.fraction := 0;
        ELSE IF (SRC2 <> QNAN) Set DE; FI;
    FI;
```

```
    Src2.exp := SRC2[62:52];
    Src2.fraction := SRC2[51:0];
    IF ((Src2.exp = 0) and (Src2.fraction !=0 )) THEN// Src2 is a denormal number
        IF DAZ THEN Src2.fraction := 0;
        ELSE IF (SRC1 <> QNAN) Set DE; FI;
    FI;

    IF    (SRC2 = QNAN) THEN{TMP[63:0] := SRC1[63:0]}
    ELSE IF(SRC1 = QNAN) THEN{TMP[63:0] := SRC2[63:0]}
    ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[63:0] := from Table 5-22
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[63:0] := from Table 5-23
    ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[63:0] := (SRC1[63:0] ≤ SRC2[63:0]) ? SRC1[63:0] : SRC2[63:0];
        01: TMP[63:0] := (SRC1[63:0] ≤ SRC2[63:0]) ? SRC2[63:0] : SRC1[63:0];
        10: TMP[63:0] := (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC1[63:0] : SRC2[63:0];
        11: TMP[63:0] := (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC2[63:0] : SRC1[63:0];
        ESAC;
    FI;

    Case(SignSelCtl[1:0])
    00: dest := (SRC1[63] << 63) OR (TMP[62:0]);// Preserve Src1 sign bit
    01: dest := TMP[63:0];// Preserve sign of compare result
    10: dest := (0 << 63) OR (TMP[62:0]);// Zero out sign bit
    11: dest := (1 << 63) OR (TMP[62:0]);// Set the sign bit
    ESAC;
    RETURN dest[63:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];
```

VRANGEPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := RangeDP (SRC1[i+63:i], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
                ELSE DEST[i+63:i] := RangeDP (SRC1[i+63:i], SRC2[i+63:i], CmpOpCtl[1:0], SignSelCtl[1:0]);
            FI;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[i+63:i] = 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±1023.

VRANGEPD zmm_dst, zmm_src, zmm_1023, 02h;

Where:

zmm_dst is the destination operand.
zmm_src is the input operand to compare against ±1023 (this is SRC1).
zmm_1023 is the reference operand, contains the value of 1023 (and this is SRC2).
IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of SRC1.sign.

In case |zmm_src| < 1023 (i.e., SRC1 is smaller than 1023 in magnitude), then its value will be written into zmm_dst. Otherwise, the value stored in zmm_dst will get the value of 1023 (received on zmm_1023, which is SRC2).
However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from zmm_src. So, even in the case of |zmm_src| ≥ 1023, the selected sign of SRC1 is kept.
Thus, if zmm_src < -1023, the result of VRANGEPD will be the minimal value of -1023 while if zmm_src > +1023, the result of VRANGE will be the maximal value of +1023.

Intel C/C++ Compiler Intrinsic Equivalent

VRANGEPD __m512d _mm512_range_pd (__m512d a, __m512d b, int imm);
VRANGEPD __m512d _mm512_range_round_pd (__m512d a, __m512d b, int imm, int sae);
VRANGEPD __m512d _mm512_mask_range_pd (__m512 ds, __mmask8 k, __m512d a, __m512d b, int imm);
VRANGEPD __m512d _mm512_mask_range_round_pd (__m512d s, __mmask8 k, __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m512d _mm512_maskz_range_pd (__mmask8 k, __m512d a, __m512d b, int imm);
VRANGEPD __m512d _mm512_maskz_range_round_pd (__mmask8 k, __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m256d _mm256_range_pd (__m256d a, __m256d b, int imm);
VRANGEPD __m256d _mm256_mask_range_pd (__m256d s, __mmask8 k, __m256d a, __m256d b, int imm);
VRANGEPD __m256d _mm256_maskz_range_pd (__mmask8 k, __m256d a, __m256d b, int imm);
VRANGEPD __m128d _mm_range_pd (__m128 a, __m128d b, int imm);
VRANGEPD __m128d _mm_mask_range_pd (__m128 s, __mmask8 k, __m128d a, __m128d b, int imm);
VRANGEPD __m128d _mm_maskz_range_pd (__mmask8 k, __m128d a, __m128d b, int imm);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

See Table 2-48, "Type E2 Class Exception Conditions."

VRANGEPS—Range Restriction Calculation for Packed Pairs of Float32 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 50 /r ib VRANGEPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Calculate four RANGE operation output value from 4 pairs of single-precision floating-point values in xmm2 and xmm3/m128/m32bcst, store the results to xmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.
EVEX.256.66.0F3A.W0 50 /r ib VRANGEPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Calculate eight RANGE operation output value from 8 pairs of single-precision floating-point values in ymm2 and ymm3/m256/m32bcst, store the results to ymm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.
EVEX.512.66.0F3A.W0 50 /r ib VRANGEPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}, imm8	A	V/V	AVX512DQ OR AVX10.1	Calculate 16 RANGE operation output value from 16 pairs of single-precision floating-point values in zmm2 and zmm3/m512/m32bcst, store the results to zmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

This instruction calculates 4/8/16 range operation outputs from two sets of packed input single precision floating-point values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of imm8[1:0] and imm8[3:2] are shown in Figure 5-27.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-21. If the comparison raises an IE, the sign select control (imm8[3:2]) has no effect to the range operation output; this is indicated also in Table 5-21.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar floating-point MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-22.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-23.

Operation

```
RangeSP(SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-21
    IF (SRC1=SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2=SNAN) THEN RETURN (QNAN(SRC2), set IE);

    Src1.exp := SRC1[30:23];
    Src1.fraction := SRC1[22:0];
    IF ((Src1.exp = 0 ) and (Src1.fraction != 0 )) THEN// Src1 is a denormal number
        IF DAZ THEN Src1.fraction := 0;
        ELSE IF (SRC2 <> QNAN) Set DE; FI;
    FI;
    Src2.exp := SRC2[30:23];
    Src2.fraction := SRC2[22:0];
    IF ((Src2.exp = 0 ) and (Src2.fraction != 0 )) THEN// Src2 is a denormal number
        IF DAZ THEN Src2.fraction := 0;
        ELSE IF (SRC1 <> QNAN) Set DE; FI;
    FI;

    IF    (SRC2 = QNAN) THEN{TMP[31:0] := SRC1[31:0]}
    ELSE IF(SRC1 = QNAN) THEN{TMP[31:0] := SRC2[31:0]}
    ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[31:0] := from Table 5-22
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] := from Table 5-23
    ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[31:0] := (SRC1[31:0] ≤ SRC2[31:0]) ? SRC1[31:0] : SRC2[31:0];
        01: TMP[31:0] := (SRC1[31:0] ≤ SRC2[31:0]) ? SRC2[31:0] : SRC1[31:0];
        10: TMP[31:0] := (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC1[31:0] : SRC2[31:0];
        11: TMP[31:0] := (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC2[31:0] : SRC1[31:0];
        ESAC;
    FI;
    Case(SignSelCtl[1:0])
    00: dest := (SRC1[31] << 31) OR (TMP[30:0]);// Preserve Src1 sign bit
    01: dest := TMP[31:0];// Preserve sign of compare result
    10: dest := (0 << 31) OR (TMP[30:0]);// Zero out sign bit
    11: dest := (1 << 31) OR (TMP[30:0]);// Set the sign bit
    ESAC;
    RETURN dest[31:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];
```

VRANGEPS

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := RangeSP (SRC1[i+31:i], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
                ELSE DEST[i+31:i] := RangeSP (SRC1[i+31:i], SRC2[i+31:i], CmpOpCtl[1:0], SignSelCtl[1:0]);
            FI;
```

```
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+31:i] = 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±150.

VRANGEPS zmm_dst, zmm_src, zmm_150, 02h;

Where:
zmm_dst is the destination operand.
zmm_src is the input operand to compare against ±150.
zmm_150 is the reference operand, contains the value of 150.
IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of src1.sign.

In case |zmm_src| < 150, then its value will be written into zmm_dst. Otherwise, the value stored in zmm_dst will get the value of 150 (received on zmm_150).
However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from zmm_src. So, even in the case of |zmm_src| ≥ 150, the selected sign of SRC1 is kept.
Thus, if zmm_src < -150, the result of VRANGEPS will be the minimal value of -150 while if zmm_src > +150, the result of VRANGE will be the maximal value of +150.

Intel C/C++ Compiler Intrinsic Equivalent

VRANGEPS __m512 _mm512_range_ps (__m512 a, __m512 b, int imm);
VRANGEPS __m512 _mm512_range_round_ps (__m512 a, __m512 b, int imm, int sae);
VRANGEPS __m512 _mm512_mask_range_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VRANGEPS __m512 _mm512_mask_range_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m512 _mm512_maskz_range_ps (__mmask16 k, __m512 a, __m512 b, int imm);
VRANGEPS __m512 _mm512_maskz_range_round_ps (__mmask16 k, __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m256 _mm256_range_ps (__m256 a, __m256 b, int imm);
VRANGEPS __m256 _mm256_mask_range_ps (__m256 s, __mmask8 k, __m256 a, __m256 b, int imm);
VRANGEPS __m256 _mm256_maskz_range_ps (__mmask8 k, __m256 a, __m256 b, int imm);
VRANGEPS __m128 _mm_range_ps (__m128 a, __m128 b, int imm);
VRANGEPS __m128 _mm_mask_range_ps (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRANGEPS __m128 _mm_maskz_range_ps (__mmask8 k, __m128 a, __m128 b, int imm);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

See Table 2-48, "Type E2 Class Exception Conditions."

VRANGESD—Range Restriction Calculation From a Pair of Scalar Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W1 51 /r VRANGESD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	A	V/V	AVX512DQ OR AVX10.1	Calculate a RANGE operation output value from 2 double precision floating-point values in xmm2 and xmm3/m64, store the output to xmm1 under writemask. Imm8 specifies the comparison and sign of the range operation.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

This instruction calculates a range operation output from two input double precision floating-point values in the low qword element of the first source operand (the second operand) and second source operand (the third operand). The range output is written to the low qword element of the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of imm8[1:0] and imm8[3:2] are shown in Figure 5-27.

Bits 128:63 of the destination operand are copied from the respective element of the first source operand.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-21. If the comparison raises an IE, the sign select control (imm8[3:2]) has no effect to the range operation output; this is indicated also in Table 5-21.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar floating-point MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-22.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-23.

Operation

```
RangeDP(SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-21
    IF (SRC1 = SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2 = SNAN) THEN RETURN (QNAN(SRC2), set IE);

    Src1.exp := SRC1[62:52];
    Src1.fraction := SRC1[51:0];
    IF ((Src1.exp = 0 ) and (Src1.fraction != 0)) THEN// Src1 is a denormal number
        IF DAZ THEN Src1.fraction := 0;
        ELSE IF (SRC2 <> QNAN) Set DE; FI;
    FI;

    Src2.exp := SRC2[62:52];
    Src2.fraction := SRC2[51:0];
    IF ((Src2.exp = 0) and (Src2.fraction !=0 )) THEN// Src2 is a denormal number
        IF DAZ THEN Src2.fraction := 0;
        ELSE IF (SRC1 <> QNAN) Set DE; FI;
    FI;

    IF    (SRC2 = QNAN) THEN{TMP[63:0] := SRC1[63:0]}
    ELSE IF(SRC1 = QNAN) THEN{TMP[63:0] := SRC2[63:0]}
    ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[63:0] := from Table 5-22
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[63:0] := from Table 5-23
    ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[63:0] := (SRC1[63:0] ≤ SRC2[63:0]) ? SRC1[63:0] : SRC2[63:0];
        01: TMP[63:0] := (SRC1[63:0] ≤ SRC2[63:0]) ? SRC2[63:0] : SRC1[63:0];
        10: TMP[63:0] := (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC1[63:0] : SRC2[63:0];
        11: TMP[63:0] := (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC2[63:0] : SRC1[63:0];
        ESAC;
    FI;

    Case(SignSelCtl[1:0])
    00: dest := (SRC1[63] << 63) OR (TMP[62:0]);// Preserve Src1 sign bit
    01: dest := TMP[63:0];// Preserve sign of compare result
    10: dest := (0 << 63) OR (TMP[62:0]);// Zero out sign bit
    11: dest := (1 << 63) OR (TMP[62:0]);// Set the sign bit
    ESAC;
    RETURN dest[63:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];
```

VRANGESD

```
IF k1[0] OR *no writemask*
        THEN DEST[63:0] := RangeDP (SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63:0] = 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±1023.

VRANGESD xmm_dst, xmm_src, xmm_1023, 02h;

Where:
xmm_dst is the destination operand.
xmm_src is the input operand to compare against ±1023.
xmm_1023 is the reference operand, contains the value of 1023.
IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of src1.sign.

In case |xmm_src| < 1023, then its value will be written into xmm_dst. Otherwise, the value stored in xmm_dst will get the value of 1023 (received on xmm_1023).
However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from xmm_src. So, even in the case of |xmm_src| ≥ 1023, the selected sign of SRC1 is kept.
Thus, if xmm_src < -1023, the result of VRANGEPD will be the minimal value of -1023while if xmm_src > +1023, the result of VRANGE will be the maximal value of +1023.

Intel C/C++ Compiler Intrinsic Equivalent

VRANGESD __m128d _mm_range_sd (__m128d a, __m128d b, int imm);
VRANGESD __m128d _mm_range_round_sd (__m128d a, __m128d b, int imm, int sae);
VRANGESD __m128d _mm_mask_range_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VRANGESD __m128d _mm_mask_range_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm, int sae);
VRANGESD __m128d _mm_maskz_range_sd (__mmask8 k, __m128d a, __m128d b, int imm);
VRANGESD __m128d _mm_maskz_range_round_sd (__mmask8 k, __m128d a, __m128d b, int imm, int sae);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

See Table 2-49, "Type E3 Class Exception Conditions."

VRANGESS—Range Restriction Calculation From a Pair of Scalar Float32 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W0 51 /r VRANGESS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512DQ OR AVX10.1	Calculate a RANGE operation output value from 2 single-precision floating-point values in xmm2 and xmm3/m32, store the output to xmm1 under writemask. Imm8 specifies the comparison and sign of the range operation.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction calculates a range operation output from two input single precision floating-point values in the low dword element of the first source operand (the second operand) and second source operand (the third operand). The range output is written to the low dword element of the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of imm8[1:0] and imm8[3:2] are shown in Figure 5-27.

Bits 128:31 of the destination operand are copied from the respective elements of the first source operand.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-21. If the comparison raises an IE, the sign select control (imm8[3:2]) has no effect to the range operation output; this is indicated also in Table 5-21.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar floating-point MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-22.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-23.

Operation

```
RangeSP(SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-21
    IF (SRC1=SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2=SNAN) THEN RETURN (QNAN(SRC2), set IE);

    Src1.exp := SRC1[30:23];
    Src1.fraction := SRC1[22:0];
    IF ((Src1.exp = 0 ) and (Src1.fraction != 0 )) THEN// Src1 is a denormal number
        IF DAZ THEN Src1.fraction := 0;
        ELSE IF (SRC2 <> QNAN) Set DE; FI;
    FI;
    Src2.exp := SRC2[30:23];
    Src2.fraction := SRC2[22:0];
    IF ((Src2.exp = 0 ) and (Src2.fraction != 0 )) THEN// Src2 is a denormal number
        IF DAZ THEN Src2.fraction := 0;
        ELSE IF (SRC1 <> QNAN) Set DE; FI;
    FI;

    IF    (SRC2 = QNAN) THEN{TMP[31:0] := SRC1[31:0]}
    ELSE IF(SRC1 = QNAN) THEN{TMP[31:0] := SRC2[31:0]}
    ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[31:0] := from Table 5-22
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] := from Table 5-23
    ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[31:0] := (SRC1[31:0] ≤ SRC2[31:0]) ? SRC1[31:0] : SRC2[31:0];
        01: TMP[31:0] := (SRC1[31:0] ≤ SRC2[31:0]) ? SRC2[31:0] : SRC1[31:0];
        10: TMP[31:0] := (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC1[31:0] : SRC2[31:0];
        11: TMP[31:0] := (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC2[31:0] : SRC1[31:0];
        ESAC;
    FI;
    Case(SignSelCtl[1:0])
    00: dest := (SRC1[31] << 31) OR (TMP[30:0]);// Preserve Src1 sign bit
    01: dest := TMP[31:0];// Preserve sign of compare result
    10: dest := (0 << 31) OR (TMP[30:0]);// Zero out sign bit
    11: dest := (1 << 31) OR (TMP[30:0]);// Set the sign bit
    ESAC;
    RETURN dest[31:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];
```

VRANGESS

```
IF k1[0] OR *no writemask*
        THEN DEST[31:0] := RangeSP (SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] = 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±150.

VRANGESS zmm_dst, zmm_src, zmm_150, 02h;

Where:
xmm_dst is the destination operand.
xmm_src is the input operand to compare against ±150.
xmm_150 is the reference operand, contains the value of 150.
IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of src1.sign.

In case |xmm_src| < 150, then its value will be written into zmm_dst. Otherwise, the value stored in xmm_dst will get the value of 150 (received on zmm_150).
However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from xmm_src. So, even in the case of |xmm_src| ≥ 150, the selected sign of SRC1 is kept.
Thus, if xmm_src < -150, the result of VRANGESS will be the minimal value of -150 while if xmm_src > +150, the result of VRANGE will be the maximal value of +150.

Intel C/C++ Compiler Intrinsic Equivalent

VRANGESS __m128 _mm_range_ss (__m128 a, __m128 b, int imm);
VRANGESS __m128 _mm_range_round_ss (__m128 a, __m128 b, int imm, int sae);
VRANGESS __m128 _mm_mask_range_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRANGESS __m128 _mm_mask_range_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm, int sae);
VRANGESS __m128 _mm_maskz_range_ss (__mmask8 k, __m128 a, __m128 b, int imm);
VRANGESS __m128 _mm_maskz_range_round_ss (__mmask8 k, __m128 a, __m128 b, int imm, int sae);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

See Table 2-49, "Type E3 Class Exception Conditions."

VRCP14PD—Compute Approximate Reciprocals of Packed Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 4C /r VRCP14PD xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Computes the approximate reciprocals of the packed double precision floating-point values in xmm2/m128/m64bcst and stores the results in xmm1. Under writemask.
EVEX.256.66.0F38.W1 4C /r VRCP14PD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Computes the approximate reciprocals of the packed double precision floating-point values in ymm2/m256/m64bcst and stores the results in ymm1. Under writemask.
EVEX.512.66.0F38.W1 4C /r VRCP14PD zmm1 {k1}{z}, zmm2/m512/m64bcst	A	V/V	AVX512F OR AVX10.1	Computes the approximate reciprocals of the packed double precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1. Under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocals of eight/four/two packed double precision floating-point values in the source operand (the second operand) and stores the packed double precision floating-point results in the destination operand. The maximum relative error for this approximation is less than 2^{-14}.

The source operand can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register conditionally updated according to the writemask.

The VRCP14PD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e., not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e., correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Table 5-24. VRCP14PD/VRCP14SD Special Cases

Input value	Result value	Comments
$0 \le X \le 2^{-1024}$	INF	Very small denormal
$-2^{-1024} \le X \le -0$	-INF	Very small denormal
$X > 2^{1022}$	Underflow	Up to 18 bits of fractions are returned*
$X < -2^{1022}$	-Underflow	Up to 18 bits of fractions are returned*
$X = 2^{-n}$	2^{n}	
$X = -2^{-n}$	-2^{n}	

* in this case the mantissa is shifted right by one or two bits

A numerically exact implementation of VRCP14xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP14PD ((EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+63:i] := APPROXIMATE(1.0/SRC[63:0]);
                ELSE DEST[i+63:i] := APPROXIMATE(1.0/SRC[i+63:i]);
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+63:i] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRCP14PD __m512d _mm512_rcp14_pd(__m512d a);
VRCP14PD __m512d _mm512_mask_rcp14_pd(__m512d s, __mmask8 k, __m512d a);
VRCP14PD __m512d _mm512_maskz_rcp14_pd(__mmask8 k, __m512d a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, "Type E4 Class Exception Conditions."

VRCP14PS—Compute Approximate Reciprocals of Packed Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 4C /r VRCP14PS xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Computes the approximate reciprocals of the packed single-precision floating-point values in xmm2/m128/m32bcst and stores the results in xmm1. Under writemask.
EVEX.256.66.0F38.W0 4C /r VRCP14PS ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Computes the approximate reciprocals of the packed single-precision floating-point values in ymm2/m256/m32bcst and stores the results in ymm1. Under writemask.
EVEX.512.66.0F38.W0 4C /r VRCP14PS zmm1 {k1}{z}, zmm2/m512/m32bcst	A	V/V	AVX512F OR AVX10.1	Computes the approximate reciprocals of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocals of the packed single precision floating-point values in the source operand (the second operand) and stores the packed single precision floating-point results in the destination operand (the first operand). The maximum relative error for this approximation is less than 2^{-14}.

The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated according to the writemask.

The VRCP14PS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e., not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e., correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Table 5-25. VRCP14PS/VRCP14SS Special Cases

Input value	Result value	Comments
$0 \le X \le 2^{-128}$	INF	Very small denormal
$-2^{-128} \le X \le -0$	-INF	Very small denormal
$X > 2^{126}$	Underflow	Up to 18 bits of fractions are returned[1]
$X < -2^{126}$	-Underflow	Up to 18 bits of fractions are returned[1]
$X = 2^{-n}$	2^{n}	
$X = -2^{-n}$	-2^{n}	

NOTES:

1. In this case, the mantissa is shifted right by one or two bits.

A numerically exact implementation of VRCP14xx can be found at:

https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP14PS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+31:i] := APPROXIMATE(1.0/SRC[31:0]);
                ELSE DEST[i+31:i] := APPROXIMATE(1.0/SRC[i+31:i]);
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+31:i] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRCP14PS __m512 _mm512_rcp14_ps(__m512 a);
VRCP14PS __m512 _mm512_mask_rcp14_ps(__m512 s, __mmask16 k, __m512 a);
VRCP14PS __m512 _mm512_maskz_rcp14_ps(__mmask16 k, __m512 a);
VRCP14PS __m256 _mm256_rcp14_ps(__m256 a);
VRCP14PS __m256 _mm512_mask_rcp14_ps(__m256 s, __mmask8 k, __m256 a);
VRCP14PS __m256 _mm512_maskz_rcp14_ps(__mmask8 k, __m256 a);
VRCP14PS __m128 _mm_rcp14_ps(__m128 a);
VRCP14PS __m128 _mm_mask_rcp14_ps(__m128 s, __mmask8 k, __m128 a);
VRCP14PS __m128 _mm_maskz_rcp14_ps(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, "Type E4 Class Exception Conditions."

VRCP14SD—Compute Approximate Reciprocal of Scalar Float64 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W1 4D /r VRCP14SD xmm1 {k1}{z}, xmm2, xmm3/m64	A	V/V	AVX512F OR AVX10.1	Computes the approximate reciprocal of the scalar double precision floating-point value in xmm3/m64 and stores the result in xmm1 using writemask k1. Also, upper double precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocal of the low double precision floating-point value in the second source operand (the third operand) stores the result in the low quadword element of the destination operand (the first operand) according to the writemask k1. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than 2^{-14}. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register.

The VRCP14SD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e., not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e., correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned. See Table 5-24 for special-case input values.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRCP14xx can be found at:

https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP14SD (EVEX version)

```
IF k1[0] OR *no writemask*
        THEN DEST[63:0] := APPROXIMATE(1.0/SRC2[63:0]);
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRCP14SD __m128d _mm_rcp14_sd(__m128d a, __m128d b);
VRCP14SD __m128d _mm_mask_rcp14_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VRCP14SD __m128d _mm_maskz_rcp14_sd(__mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-60, "Type E10 Class Exception Conditions."

VRCP14SS—Compute Approximate Reciprocal of Scalar Float32 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W0 4D /r VRCP14SS xmm1 {k1}{z}, xmm2, xmm3/m32	A	V/V	AVX512F OR AVX10.1	Computes the approximate reciprocal of the scalar single-precision floating-point value in xmm3/m32 and stores the results in xmm1 using writemask k1. Also, upper double precision floating-point value (bits[127:32]) from xmm2 is copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocal of the low single precision floating-point value in the second source operand (the third operand) and stores the result in the low quadword element of the destination operand (the first operand) according to the writemask k1. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than 2^{-14}. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

The VRCP14SS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e., not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e., correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned. See Table 5-25 for special-case input values.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRCP14xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP14SS (EVEX version)

```
IF k1[0] OR *no writemask*
        THEN DEST[31:0] := APPROXIMATE(1.0/SRC2[31:0]);
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRCP14SS __m128 _mm_rcp14_ss(__m128 a, __m128 b);
VRCP14SS __m128 _mm_mask_rcp14_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VRCP14SS __m128 _mm_maskz_rcp14_ss(__mmask8 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-60, "Type E10 Class Exception Conditions."

VRCPPH—Compute Reciprocals of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP6.W0 4C /r VRCPPH xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Compute the approximate reciprocals of packed FP16 values in xmm2/m128/m16bcst and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 4C /r VRCPPH ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Compute the approximate reciprocals of packed FP16 values in ymm2/m256/m16bcst and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 4C /r VRCPPH zmm1{k1}{z}, zmm2/m512/m16bcst	A	V/V	AVX512_FP16 OR AVX10.1	Compute the approximate reciprocals of packed FP16 values in zmm2/m512/m16bcst and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocals of 8/16/32 packed FP16 values in the source operand (the second operand) and stores the packed FP16 results in the destination operand. The maximum relative error for this approximation is less than $2^{-11} + 2^{-14}$.

For special cases, see Table 5-26.

Table 5-26. VRCPPH/VRCPSH Special Cases

Input Value	Result Value	Comments
$0 \le X \le 2^{-16}$	INF	Very small denormal
$-2^{-16} \le X \le -0$	–INF	Very small denormal
$X > +\infty$	+0	
$X < -\infty$	–0	
$X = 2^{-n}$	2^{n}	
$X = -2^{-n}$	-2^{n}	

Operation

VRCPPH dest{k1}, src

```
VL = 128, 256 or 512
KL := VL/16

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.fp16[0]
        ELSE:
            tsrc := src.fp16[i]
        DEST.fp16[i] := APPROXIMATE(1.0 / tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[i] := 0
    //else DEST.fp16[i] remains unchanged
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRCPPH __m128h _mm_mask_rcp_ph (__m128h src, __mmask8 k, __m128h a);
VRCPPH __m128h _mm_maskz_rcp_ph (__mmask8 k, __m128h a);
VRCPPH __m128h _mm_rcp_ph (__m128h a);
VRCPPH __m256h _mm256_mask_rcp_ph (__m256h src, __mmask16 k, __m256h a);
VRCPPH __m256h _mm256_maskz_rcp_ph (__mmask16 k, __m256h a);
VRCPPH __m256h _mm256_rcp_ph (__m256h a);
VRCPPH __m512h _mm512_mask_rcp_ph (__m512h src, __mmask32 k, __m512h a);
VRCPPH __m512h _mm512_maskz_rcp_ph (__mmask32 k, __m512h a);
VRCPPH __m512h _mm512_rcp_ph (__m512h a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

VRCPSH—Compute Reciprocal of Scalar FP16 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.MAP6.W0 4D /r VRCPSH xmm1{k1}{z}, xmm2, xmm3/m16	A	V/V	AVX512_FP16 OR AVX10.1	Compute the approximate reciprocal of the low FP16 value in xmm3/m16 and store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocal of the low FP16 value in the second source operand (the third operand) and stores the result in the low word element of the destination operand (the first operand) according to the writemask k1. Bits 127:16 of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than $2^{-11} + 2^{-14}$.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

For special cases, see Table 5-26.

Operation

VRCPSH dest{k1}, src1, src2

```
IF k1[0] or *no writemask*:
    DEST.fp16[0] := APPROXIMATE(1.0 / src2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
//else DEST.fp16[0] remains unchanged

DEST[127:16] := src1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRCPSH __m128h _mm_mask_rcp_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VRCPSH __m128h _mm_maskz_rcp_sh (__mmask8 k, __m128h a, __m128h b);
VRCPSH __m128h _mm_rcp_sh (__m128h a, __m128h b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-60, “Type E10 Class Exception Conditions.”

VREDUCEPD—Perform Reduction Transformation on Packed Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 56 /r ib VREDUCEPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Perform reduction transformation on packed double precision floating-point values in xmm2/m128/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask k1.
EVEX.256.66.0F3A.W1 56 /r ib VREDUCEPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Perform reduction transformation on packed double precision floating-point values in ymm2/m256/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register under writemask k1.
EVEX.512.66.0F3A.W1 56 /r ib VREDUCEPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, imm8	A	V/V	AVX512DQ OR AVX10.1	Perform reduction transformation on double precision floating-point values in zmm2/m512/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Perform reduction transformation of the packed binary encoded double precision floating-point values in the source operand (the second operand) and store the reduced results in binary floating-point format to the destination operand (the first operand) under the writemask k1.

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary floating-point source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

dest = src – (ROUND(2^M*src))*2^{-M};

where "Round()" treats "src", "2^M", and their product as binary floating-point numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering src= 2^p*man2,
where 'man2' is the normalized significand and 'p' is the unbiased exponent

Then if RC = RNE: 0<=|Reduced Result|<=2^{p-M-1}

Then if RC ≠ RNE: 0<=|Reduced Result|<2^{p-M}

This instruction might end up with a precision exception set. However, in case of SPE set (i.e., Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

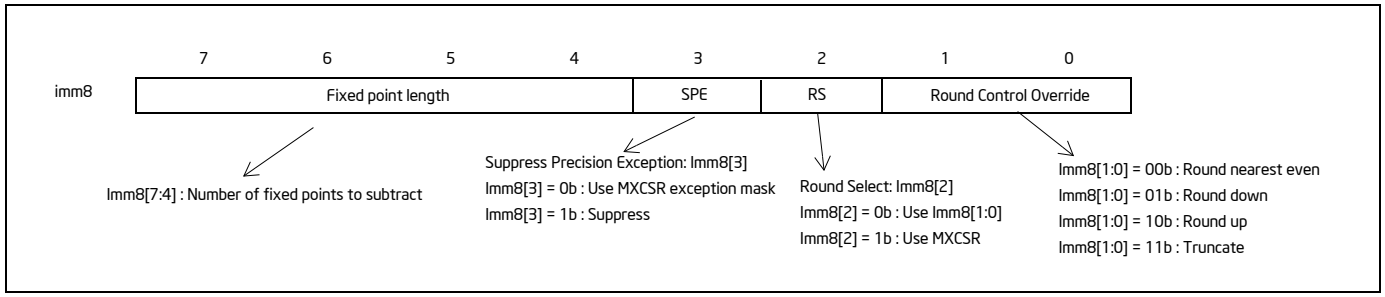


Figure 5-28. Imm8 Controls for VREDUCEPD/SD/PS/SS

Handling of special case of input values are listed in Table 5-27.

Table 5-27. VREDUCEPD/SD/PS/SS Special Cases

	Round Mode	Returned value
$\|Src1\| < 2^{-M-1}$	RNE	Src1
$\|Src1\| < 2^{-M}$	RPI, Src1 > 0	Round (Src1-2^{-M}) *
	RPI, Src1 ≤ 0	Src1
	RNI, Src1 ≥ 0	Src1
	RNI, Src1 < 0	Round (Src1+2^{-M}) *
Src1 = ±0, or Dest = ±0 (Src1!=INF)	NOT RNI	+0.0
	RNI	-0.0
Src1 = ±INF	any	+0.0
Src1= ±NAN	n/a	QNaN(Src1)

* Round control = (imm8.MS1)? MXCSR.RC: imm8.RC

Operation

```
ReduceArgumentDP(SRC[63:0], imm8[7:0])
{
    // Check for NaN
    IF (SRC [63:0] = NAN) THEN
        RETURN (Convert SRC[63:0] to QNaN); FI;
    M := imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
    RC := imm8[1:0];// Round Control for ROUND() operation
    RC source := imm[2];
    SPE := imm[3];// Suppress Precision Exception
    TMP[63:0] := 2^-M *{ROUND(2^M*SRC[63:0], SPE, RC_source, RC)}; // ROUND() treats SRC and 2^M as standard binary FP values
    TMP[63:0] := SRC[63:0] – TMP[63:0]; // subtraction under the same RC,SPE controls
    RETURN TMP[63:0]; // binary encoded FP with biased exponent and normalized significand
}
```

VREDUCEPD

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b == 1) AND (SRC *is memory*)
                THEN DEST[i+63:i] := ReduceArgumentDP(SRC[63:0], imm8[7:0]);
                ELSE DEST[i+63:i] := ReduceArgumentDP(SRC[i+63:i], imm8[7:0]);
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+63:i] = 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VREDUCEPD __m512d _mm512_mask_reduce_pd(__m512d a, int imm, int sae)
VREDUCEPD __m512d _mm512_mask_reduce_pd(__m512d s, __mmask8 k, __m512d a, int imm, int sae)
VREDUCEPD __m512d _mm512_maskz_reduce_pd(__mmask8 k, __m512d a, int imm, int sae)
VREDUCEPD __m256d _mm256_mask_reduce_pd(__m256d a, int imm)
VREDUCEPD __m256d _mm256_mask_reduce_pd(__m256d s, __mmask8 k, __m256d a, int imm)
VREDUCEPD __m256d _mm256_maskz_reduce_pd(__mmask8 k, __m256d a, int imm)
VREDUCEPD __m128d _mm_mask_reduce_pd(__m128d a, int imm)
VREDUCEPD __m128d _mm_mask_reduce_pd(__m128d s, __mmask8 k, __m128d a, int imm)
VREDUCEPD __m128d _mm_maskz_reduce_pd(__mmask8 k, __m128d a, int imm)

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-48, "Type E2 Class Exception Conditions."
Additionally:

#UD	If EVEX.vvvv != 1111B.

VREDUCEPH—Perform Reduction Transformation on Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.0F3A.W0 56 /r /ib VREDUCEPH xmm1{k1}{z}, xmm2/m128/m16bcst, imm8	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Perform reduction transformation on packed FP16 values in xmm2/m128/m16bcst by subtracting a number of fraction bits specified by the imm8 field. Store the result in xmm1 subject to writemask k1.
EVEX.256.NP.0F3A.W0 56 /r /ib VREDUCEPH ymm1{k1}{z}, ymm2/m256/m16bcst, imm8	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Perform reduction transformation on packed FP16 values in ymm2/m256/m16bcst by subtracting a number of fraction bits specified by the imm8 field. Store the result in ymm1 subject to writemask k1.
EVEX.512.NP.0F3A.W0 56 /r /ib VREDUCEPH zmm1{k1}{z}, zmm2/m512/m16bcst {sae}, imm8	A	V/V	AVX512_FP16 OR AVX10.1	Perform reduction transformation on packed FP16 values in zmm2/m512/m16bcst by subtracting a number of fraction bits specified by the imm8 field. Store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8 (r)	N/A

Description

This instruction performs a reduction transformation of the packed binary encoded FP16 values in the source operand (the second operand) and store the reduced results in binary FP format to the destination operand (the first operand) under the writemask k1.

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4]. Specifically, the reduction transformation can be expressed as:

dest = src − (ROUND(2^M * src)) * 2^{-M}

where ROUND() treats src, 2^M, and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering src = 2^p * man2, where 'man2' is the normalized significand and 'p' is the unbiased exponent.

Then if RC=RNE: 0 ≤ |ReducedResult| ≤ 2^{-M-1}.

Then if RC ≠ RNE: 0 ≤ |ReducedResult| < 2^{-M}.

This instruction might end up with a precision exception set. However, in case of SPE set (i.e., Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

This instruction may generate tiny non-zero result. If it does so, it does not report underflow exception, even if underflow exceptions are unmasked (UM flag in MXCSR register is 0).

For special cases, see Table 5-28.

Table 5-28. VREDUCEPH/VREDUCESH Special Cases

Input value	Round Mode	Returned Value
$\|Src1\| < 2^{-M-1}$	RNE	Src1
$\|Src1\| < 2^{-M}$	RU, Src1 > 0	Round(Src1 – 2^{-M})[1]
	RU, Src1 ≤ 0	Src1
	RD, Src1 ≥ 0	Src1
	RD, Src1 < 0	Round(Src1 + 2^{-M})
Src1 = ±0 or Dest = ±0 (Src1 ≠ ∞)	NOT RD	+0.0
	RD	–0.0
Src1 = ±∞	Any	+0.0
Src1 = ±NAN	Any	QNaN (Src1)

NOTES:

1. The Round(.) function uses rounding controls specified by (imm8[2]? MXCSR.RC: imm8[1:0]).

Operation

```
def reduce_fp16(src, imm8):
    nan := (src.exp = 0x1F) and (src.fraction != 0)
    if nan:
        return QNAN(src)
    m := imm8[7:4]
    rc := imm8[1:0]
    rc_source := imm8[2]
    spe := imm[3] // suppress precision exception
    tmp := 2^(-m) * ROUND(2^m * src, spe, rc_source, rc)
    tmp := src - tmp // using same RC, SPE controls
    return tmp
```

VREDUCEPH dest{k1}, src, imm8

```
VL = 128, 256 or 512
KL := VL/16

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.fp16[0]
        ELSE:
            tsrc := src.fp16[i]
        DEST.fp16[i] := reduce_fp16(tsrc, imm8)
    ELSE IF *zeroing*:
        DEST.fp16[i] := 0
    //else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VREDUCEPH __m128h _mm_mask_reduce_ph (__m128h src, __mmask8 k, __m128h a, int imm8);
VREDUCEPH __m128h _mm_maskz_reduce_ph (__mmask8 k, __m128h a, int imm8);
VREDUCEPH __m128h _mm_reduce_ph (__m128h a, int imm8);
VREDUCEPH __m256h _mm256_mask_reduce_ph (__m256h src, __mmask16 k, __m256h a, int imm8);
VREDUCEPH __m256h _mm256_maskz_reduce_ph (__mmask16 k, __m256h a, int imm8);
VREDUCEPH __m256h _mm256_reduce_ph (__m256h a, int imm8);
VREDUCEPH __m512h _mm512_mask_reduce_ph (__m512h src, __mmask32 k, __m512h a, int imm8);
VREDUCEPH __m512h _mm512_maskz_reduce_ph (__mmask32 k, __m512h a, int imm8);
VREDUCEPH __m512h _mm512_reduce_ph (__m512h a, int imm8);
VREDUCEPH __m512h _mm512_mask_reduce_round_ph (__m512h src, __mmask32 k, __m512h a, int imm8, const int sae);
VREDUCEPH __m512h _mm512_maskz_reduce_round_ph (__mmask32 k, __m512h a, int imm8, const int sae);
VREDUCEPH __m512h _mm512_reduce_round_ph (__m512h a, int imm8, const int sae);

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

EVEX-encoded instruction, see Table 2-48, "Type E2 Class Exception Conditions."

VREDUCEPS—Perform Reduction Transformation on Packed Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 56 /r ib VREDUCEPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Perform reduction transformation on packed single-precision floating-point values in xmm2/m128/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask k1.
EVEX.256.66.0F3A.W0 56 /r ib VREDUCEPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Perform reduction transformation on packed single-precision floating-point values in ymm2/m256/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register under writemask k1.
EVEX.512.66.0F3A.W0 56 /r ib VREDUCEPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}, imm8	A	V/V	AVX512DQ OR AVX10.1	Perform reduction transformation on packed single-precision floating-point values in zmm2/m512/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Perform reduction transformation of the packed binary encoded single precision floating-point values in the source operand (the second operand) and store the reduced results in binary floating-point format to the destination operand (the first operand) under the writemask k1.

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary floating-point source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

dest = src – (ROUND(2^M*src))*2^{-M};

where "Round()" treats "src", "2^M", and their product as binary floating-point numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering src= 2^p*man2,
where 'man2' is the normalized significand and 'p' is the unbiased exponent

Then if RC = RNE: 0<=|Reduced Result|<=2^{p-M-1}

Then if RC ≠ RNE: 0<=|Reduced Result|<2^{p-M}

This instruction might end up with a precision exception set. However, in case of SPE set (i.e., Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Handling of special case of input values are listed in Table 5-27.

Operation

```
ReduceArgumentSP(SRC[31:0], imm8[7:0])
{
    // Check for NaN
    IF (SRC [31:0] = NAN) THEN
        RETURN (Convert SRC[31:0] to QNaN); FI
    M := imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
    RC := imm8[1:0];// Round Control for ROUND() operation
    RC source := imm[2];
    SPE := imm[3];// Suppress Precision Exception
    TMP[31:0] := 2^-M *{ROUND(2^M*SRC[31:0], SPE, RC_source, RC)}; // ROUND() treats SRC and 2^M as standard binary FP values
    TMP[31:0] := SRC[31:0] – TMP[31:0]; // subtraction under the same RC,SPE controls
RETURN TMP[31:0]; // binary encoded FP with biased exponent and normalized significand
}
```

VREDUCEPS

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b == 1) AND (SRC *is memory*)
                THEN DEST[i+31:i] := ReduceArgumentSP(SRC[31:0], imm8[7:0]);
                ELSE DEST[i+31:i] := ReduceArgumentSP(SRC[i+31:i], imm8[7:0]);
            FI;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[i+31:i] = 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VREDUCEPS __m512 _mm512_mask_reduce_ps(__m512 a, int imm, int sae)
VREDUCEPS __m512 _mm512_mask_reduce_ps(__m512 s, __mmask16 k, __m512 a, int imm, int sae)
VREDUCEPS __m512 _mm512_maskz_reduce_ps(__mmask16 k, __m512 a, int imm, int sae)
VREDUCEPS __m256 _mm256_mask_reduce_ps(__m256 a, int imm)
VREDUCEPS __m256 _mm256_mask_reduce_ps(__m256 s, __mmask8 k, __m256 a, int imm)
VREDUCEPS __m256 _mm256_maskz_reduce_ps(__mmask8 k, __m256 a, int imm)
VREDUCEPS __m128 _mm_mask_reduce_ps(__m128 a, int imm)
VREDUCEPS __m128 _mm_mask_reduce_ps(__m128 s, __mmask8 k, __m128 a, int imm)
VREDUCEPS __m128 _mm_maskz_reduce_ps(__mmask8 k, __m128 a, int imm)

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-48, "Type E2 Class Exception Conditions"; additionally:

#UD	If EVEX.vvvv != 1111B.

VREDUCESD—Perform a Reduction Transformation on a Scalar Float64 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W1 57 VREDUCESD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8/r	A	V/V	AVX512DQ OR AVX10.1	Perform a reduction transformation on a scalar double precision floating-point value in xmm3/m64 by subtracting a number of fraction bits specified by the imm8 field. Also, upper double precision floating-point value (bits[127:64]) from xmm2 are copied to xmm1[127:64]. Stores the result in xmm1 register.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Perform a reduction transformation of the binary encoded double precision floating-point value in the low qword element of the second source operand (the third operand) and store the reduced result in binary floating-point format to the low qword element of the destination operand (the first operand) under the writemask k1. Bits 127:64 of the destination operand are copied from respective qword elements of the first source operand (the second operand).

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary floating-point source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

dest = src – (ROUND(2^M*src))*2^{-M};

where "Round()" treats "src", "2^M", and their product as binary floating-point numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering src= 2^p*man2,
where 'man2' is the normalized significand and 'p' is the unbiased exponent

Then if RC = RNE: 0<=|Reduced Result|<=2^{p-M-1}

Then if RC ≠ RNE: 0<=|Reduced Result|<2^{p-M}

This instruction might end up with a precision exception set. However, in case of SPE set (i.e., Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

The operation is write masked.

Handling of special case of input values are listed in Table 5-27.

Operation

```
ReduceArgumentDP(SRC[63:0], imm8[7:0])
{
    // Check for NaN
    IF (SRC [63:0] = NAN) THEN
        RETURN (Convert SRC[63:0] to QNaN); FI;
    M := imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
    RC := imm8[1:0];// Round Control for ROUND() operation
    RC source := imm[2];
    SPE := imm[3];// Suppress Precision Exception
    TMP[63:0] := 2^-M *{ROUND(2^M*SRC[63:0], SPE, RC_source, RC)}; // ROUND() treats SRC and 2^M as standard binary FP values
    TMP[63:0] := SRC[63:0] – TMP[63:0]; // subtraction under the same RC,SPE controls
    RETURN TMP[63:0]; // binary encoded FP with biased exponent and normalized significand
}
```

VREDUCESD

```
IF k1[0] or *no writemask*
    THEN     DEST[63:0] := ReduceArgumentDP(SRC2[63:0], imm8[7:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] = 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VREDUCESD __m128d _mm_mask_reduce_sd(__m128d a, __m128d b, int imm, int sae)
VREDUCESD __m128d _mm_mask_reduce_sd(__m128d s, __mmask16 k, __m128d a, __m128d b, int imm, int sae)
VREDUCESD __m128d _mm_maskz_reduce_sd(__mmask16 k, __m128d a, __m128d b, int imm, int sae)

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-49, "Type E3 Class Exception Conditions."

VREDUCESH—Perform Reduction Transformation on Scalar FP16 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.NP.0F3A.W0 57 /r /ib VREDUCESH xmm1{k1}{z}, xmm2, xmm3/m16 {sae}, imm8	A	V/V	AVX512_FP16 OR AVX10.1	Perform a reduction transformation on the low binary encoded FP16 value in xmm3/m16 by subtracting a number of fraction bits specified by the imm8 field. Store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

This instruction performs a reduction transformation of the low binary encoded FP16 value in the source operand (the second operand) and store the reduced result in binary FP format to the low element of the destination operand (the first operand) under the writemask k1. For further details see the description of VREDUCEPH.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

This instruction might end up with a precision exception set. However, in case of SPE set (i.e., Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

This instruction may generate tiny non-zero result. If it does so, it does not report underflow exception, even if underflow exceptions are unmasked (UM flag in MXCSR register is 0).

For special cases, see Table 5-28.

Operation

VREDUCESH dest{k1}, src, imm8

```
IF k1[0] or *no writemask*:
    dest.fp16[0] := reduce_fp16(src2.fp16[0], imm8)     // see VREDUCEPH
ELSE IF *zeroing*:
    dest.fp16[0] := 0
//else dest.fp16[0] remains unchanged

DEST[127:16] := src1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VREDUCESH __m128h _mm_mask_reduce_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int imm8, const int sae);
VREDUCESH __m128h _mm_maskz_reduce_round_sh (__mmask8 k, __m128h a, __m128h b, int imm8, const int sae);
VREDUCESH __m128h _mm_reduce_round_sh (__m128h a, __m128h b, int imm8, const int sae);
VREDUCESH __m128h _mm_mask_reduce_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int imm8);
VREDUCESH __m128h _mm_maskz_reduce_sh (__mmask8 k, __m128h a, __m128h b, int imm8);
VREDUCESH __m128h _mm_reduce_sh (__m128h a, __m128h b, int imm8);

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VREDUCESS—Perform a Reduction Transformation on a Scalar Float32 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W0 57 /r /ib VREDUCESS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512DQ OR AVX10.1	Perform a reduction transformation on a scalar single-precision floating-point value in xmm3/m32 by subtracting a number of fraction bits specified by the imm8 field. Also, upper single-precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32]. Stores the result in xmm1 register.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Perform a reduction transformation of the binary encoded single precision floating-point value in the low dword element of the second source operand (the third operand) and store the reduced result in binary floating-point format to the low dword element of the destination operand (the first operand) under the writemask k1. Bits 127:32 of the destination operand are copied from respective dword elements of the first source operand (the second operand).

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary floating-point source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

dest = src – (ROUND(2^M*src))*2^{-M};

where "Round()" treats "src", "2^M", and their product as binary floating-point numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering src= 2^p*man2,
where 'man2' is the normalized significand and 'p' is the unbiased exponent

Then if RC = RNE: 0<=|Reduced Result|<=2^{p-M-1}

Then if RC ≠ RNE: 0<=|Reduced Result|<2^{p-M}

This instruction might end up with a precision exception set. However, in case of SPE set (i.e., Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

Handling of special case of input values are listed in Table 5-27.

Operation

```
ReduceArgumentSP(SRC[31:0], imm8[7:0])
{
    // Check for NaN
    IF (SRC [31:0] = NAN) THEN
        RETURN (Convert SRC[31:0] to QNaN); FI
    M := imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
    RC := imm8[1:0];// Round Control for ROUND() operation
    RC source := imm[2];
    SPE := imm[3];// Suppress Precision Exception
    TMP[31:0] := 2^-M *{ROUND(2^M*SRC[31:0], SPE, RC_source, RC)}; // ROUND() treats SRC and 2^M as standard binary FP values
    TMP[31:0] := SRC[31:0] – TMP[31:0]; // subtraction under the same RC,SPE controls
RETURN TMP[31:0]; // binary encoded FP with biased exponent and normalized significand
}
```

VREDUCESS

```
IF k1[0] or *no writemask*
    THEN     DEST[31:0] := ReduceArgumentSP(SRC2[31:0], imm8[7:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] = 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VREDUCESS __m128 _mm_mask_reduce_ss(__m128 a, __m128 b, int imm, int sae)
VREDUCESS __m128 _mm_mask_reduce_ss(__m128 s, __mmask16 k, __m128 a, __m128 b, int imm, int sae)
VREDUCESS __m128 _mm_maskz_reduce_ss(__mmask16 k, __m128 a, __m128 b, int imm, int sae)

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-49, "Type E3 Class Exception Conditions."

VRNDSCALEPD—Round Packed Float64 Values to Include a Given Number of Fraction Bits

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 09 /r ib VRNDSCALEPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rounds packed double precision floating-point values in xmm2/m128/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register. Under writemask.
EVEX.256.66.0F3A.W1 09 /r ib VRNDSCALEPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rounds packed double precision floating-point values in ymm2/m256/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register. Under writemask.
EVEX.512.66.0F3A.W1 09 /r ib VRNDSCALEPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, imm8	A	V/V	AVX512F OR AVX10.1	Rounds packed double precision floating-point values in zmm2/m512/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Round the double precision floating-point values in the source operand by the rounding mode specified in the immediate operand (see Figure 5-29) and places the result in the destination operand.

The destination operand (the first operand) is a ZMM/YMM/XMM register conditionally updated according to the writemask. The source operand (the second operand) can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a double precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to `1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation on each data element for VRNDSCALEPD is

ROUND(x) = 2^{-M}*Round_to_INT(x*2^{M}, round_ctrl),

round_ctrl = imm[3:0];

M=imm[7:4];

The operation of x*2^{M} is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

VRNDSCALEPD is a more general form of the VEX-encoded VROUNDPD instruction. In VROUNDPD, the formula of the operation on each element is

ROUND(x) = Round_to_INT(x, round_ctrl),

round_ctrl = imm[3:0];

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

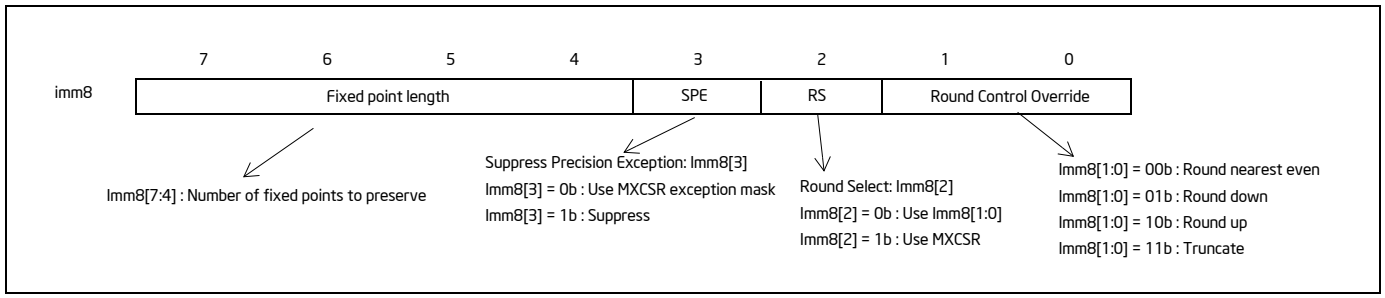


Figure 5-29. Imm8 Controls for VRNDSCALEPD/SD/PS/SS

Handling of special case of input values are listed in Table 5-29.

Table 5-29. VRNDSCALEPD/SD/PS/SS Special Cases

	Returned value
Src1=±inf	Src1
Src1=±NAN	Src1 converted to QNAN
Src1=±0	Src1

Operation

```
RoundToIntegerDP(SRC[63:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction := MXCSR:RC        ; get round control from MXCSR
    else
        rounding_direction := imm8[1:0]       ; get round control from imm8[1:0]
    FI
    M := imm8[7:4]            ; get the scaling factor

    case (rounding_direction)
    00: TMP[63:0] := round_to_nearest_even_integer(2^M*SRC[63:0])
    01: TMP[63:0] := round_to_equal_or_smaller_integer(2^M*SRC[63:0])
    10: TMP[63:0] := round_to_equal_or_larger_integer(2^M*SRC[63:0])
    11: TMP[63:0] := round_to_nearest_smallest_magnitude_integer(2^M*SRC[63:0])
    ESAC

    Dest[63:0] := 2^-M* TMP[63:0]             ; scale down back to 2^-M

    if (imm8[3] = 0) Then    ; check SPE
        if (SRC[63:0] != Dest[63:0]) Then       ; check precision lost
            set_precision()                     ; set #PE
        FI;
    FI;
```

```
    return(Dest[63:0])
}
```

VRNDSCALEPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF *src is a memory operand*
    THEN TMP_SRC := BROADCAST64(SRC, VL, k1)
    ELSE TMP_SRC := SRC
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := RoundToIntegerDP((TMP_SRC[i+63:i], imm8[7:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+63:i] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VRNDSCALEPD __m512d _mm512_roundscale_pd( __m512d a, int imm);
VRNDSCALEPD __m512d _mm512_roundscale_round_pd( __m512d a, int imm, int sae);
VRNDSCALEPD __m512d _mm512_mask_roundscale_pd(__m512d s, __mmask8 k, __m512d a, int imm);
VRNDSCALEPD __m512d _mm512_mask_roundscale_round_pd(__m512d s, __mmask8 k, __m512d a, int imm, int sae);
VRNDSCALEPD __m512d _mm512_maskz_roundscale_pd( __mmask8 k, __m512d a, int imm);
VRNDSCALEPD __m512d _mm512_maskz_roundscale_round_pd( __mmask8 k, __m512d a, int imm, int sae);
VRNDSCALEPD __m256d _mm256_roundscale_pd( __m256d a, int imm);
VRNDSCALEPD __m256d _mm256_mask_roundscale_pd(__m256d s, __mmask8 k, __m256d a, int imm);
VRNDSCALEPD __m256d _mm256_maskz_roundscale_pd( __mmask8 k, __m256d a, int imm);
VRNDSCALEPD __m128d _mm_roundscale_pd( __m128d a, int imm);
VRNDSCALEPD __m128d _mm_mask_roundscale_pd(__m128d s, __mmask8 k, __m128d a, int imm);
VRNDSCALEPD __m128d _mm_maskz_roundscale_pd( __mmask8 k, __m128d a, int imm);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-48, "Type E2 Class Exception Conditions."

VRNDSCALEPH—Round Packed FP16 Values to Include a Given Number of Fraction Bits

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.0F3A.W0 08 /r /ib VRNDSCALEPH xmm1{k1}{z}, xmm2/m128/m16bcst, imm8	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Round packed FP16 values in xmm2/m128/m16bcst to a number of fraction bits specified by the imm8 field. Store the result in xmm1 subject to writemask k1.
EVEX.256.NP.0F3A.W0 08 /r /ib VRNDSCALEPH ymm1{k1}{z}, ymm2/m256/m16bcst, imm8	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Round packed FP16 values in ymm2/m256/m16bcst to a number of fraction bits specified by the imm8 field. Store the result in ymm1 subject to writemask k1.
EVEX.512.NP.0F3A.W0 08 /r /ib VRNDSCALEPH zmm1{k1}{z}, zmm2/m512/m16bcst {sae}, imm8	A	V/V	AVX512_FP16 OR AVX10.1	Round packed FP16 values in zmm2/m512/m16bcst to a number of fraction bits specified by the imm8 field. Store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8 (r)	N/A

Description

This instruction rounds the FP16 values in the source operand by the rounding mode specified in the immediate operand (see Table 5-30) and places the result in the destination operand. The destination operand is conditionally updated according to the writemask.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result), and returns the result as an FP16 value.

Note that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation. Three bit fields are defined and shown in Table 5-30, "Imm8 Controls for VRNDSCALEPH/VRNDSCALESH." Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control, and bits 1:0 specify a non-sticky rounding-mode value.

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN.

The sign of the result of this instruction is preserved, including the sign of zero. Special cases are described in Table 5-31.

The formula of the operation on each data element for VRNDSCALEPH is

ROUND(x) = 2^{-M} *Round_to_INT(x * 2^{M}, round_ctrl),

round_ctrl = imm[3:0];

M=imm[7:4];

The operation of x * 2^{M} is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

If this instruction encoding's SPE bit (bit 3) in the immediate operand is 1, VRNDSCALEPH can set MXCSR.UE without MXCSR.PE.

EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Table 5-30. Imm8 Controls for VRNDSCALEPH/VRNDSCALESH

Imm8 Bits	Description
imm8[7:4]	Number of fixed points to preserve.
imm8[3]	Suppress Precision Exception (SPE) 0b00: Implies use of MXCSR exception mask. 0b01: Implies suppress.
imm8[2]	Round Select (RS) 0b00: Implies use of imm8[1:0]. 0b01: Implies use of MXCSR.
imm8[1:0]	Round Control Override: 0b00: Round nearest even. 0b01: Round down. 0b10: Round up. 0b11: Truncate.

Table 5-31. VRNDSCALEPH/VRNDSCALESH Special Cases

Input Value	Returned Value
Src1 = ±∞	Src1
Src1 = ±NaN	Src1 converted to QNaN
Src1 = ±0	Src1

Operation

```
def round_fp16_to_integer(src, imm8):
    if imm8[2] = 1:
        rounding_direction := MXCSR.RC
    else:
        rounding_direction := imm8[1:0]
    m := imm8[7:4] // scaling factor

    tsrc1 := 2^m * src

    if rounding_direction = 0b00:
        tmp := round_to_nearest_even_integer(trc1)
    else if rounding_direction = 0b01:
        tmp := round_to_equal_or_smaller_integer(trc1)
    else if rounding_direction = 0b10:
        tmp := round_to_equal_or_larger_integer(trc1)
    else if rounding_direction = 0b11:
        tmp := round_to_smallest_magnitude_integer(trc1)

    dst := 2^(-m) * tmp

    if imm8[3]==0: // check SPE
        if src != dst:
            MXCSR.PE := 1
    return dst
```

VRNDSCALEPH dest{k1}, src, imm8

```
VL = 128, 256 or 512
KL := VL/16

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.fp16[0]
        ELSE:
            tsrc := src.fp16[i]
        DEST.fp16[i] := round_fp16_to_integer(tsrc, imm8)
    ELSE IF *zeroing*:
        DEST.fp16[i] := 0
    //else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRNDSCALEPH __m128h _mm_mask_roundscale_ph (__m128h src, __mmask8 k, __m128h a, int imm8);
VRNDSCALEPH __m128h _mm_maskz_roundscale_ph (__mmask8 k, __m128h a, int imm8);
VRNDSCALEPH __m128h _mm_roundscale_ph (__m128h a, int imm8);
VRNDSCALEPH __m256h _mm256_mask_roundscale_ph (__m256h src, __mmask16 k, __m256h a, int imm8);
VRNDSCALEPH __m256h _mm256_maskz_roundscale_ph (__mmask16 k, __m256h a, int imm8);
VRNDSCALEPH __m256h _mm256_roundscale_ph (__m256h a, int imm8);
VRNDSCALEPH __m512h _mm512_mask_roundscale_ph (__m512h src, __mmask32 k, __m512h a, int imm8);
VRNDSCALEPH __m512h _mm512_maskz_roundscale_ph (__mmask32 k, __m512h a, int imm8);
VRNDSCALEPH __m512h _mm512_roundscale_ph (__m512h a, int imm8);
VRNDSCALEPH __m512h _mm512_mask_roundscale_round_ph (__m512h src, __mmask32 k, __m512h a, int imm8, const int sae);
VRNDSCALEPH __m512h _mm512_maskz_roundscale_round_ph (__mmask32 k, __m512h a, int imm8, const int sae);
VRNDSCALEPH __m512h _mm512_roundscale_round_ph (__m512h a, int imm8, const int sae);

SIMD Floating-Point Exceptions

Invalid, Underflow, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

EVEX-encoded instruction, see Table 2-48, "Type E2 Class Exception Conditions."

VRNDSCALEPS—Round Packed Float32 Values to Include a Given Number of Fraction Bits

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 08 /r ib VRNDSCALEPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rounds packed single-precision floating-point values in xmm2/m128/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register. Under writemask.
EVEX.256.66.0F3A.W0 08 /r ib VRNDSCALEPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rounds packed single-precision floating-point values in ymm2/m256/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register. Under writemask.
EVEX.512.66.0F3A.W0 08 /r ib VRNDSCALEPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}, imm8	A	V/V	AVX512F OR AVX10.1	Rounds packed single-precision floating-point values in zmm2/m512/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register using writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Round the single precision floating-point values in the source operand by the rounding mode specified in the immediate operand (see Figure 5-29) and places the result in the destination operand.

The destination operand (the first operand) is a ZMM register conditionally updated according to the writemask. The source operand (the second operand) can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a single precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to `1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation on each data element for VRNDSCALEPS is

ROUND(x) = 2^{-M}*Round_to_INT(x*2^{M}, round_ctrl),

round_ctrl = imm[3:0];

M=imm[7:4];

The operation of x*2^{M} is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

VRNDSCALEPS is a more general form of the VEX-encoded VROUNDPS instruction. In VROUNDPS, the formula of the operation on each element is

ROUND(x) = Round_to_INT(x, round_ctrl),

round_ctrl = imm[3:0];

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Handling of special case of input values are listed in Table 5-29.

Operation

```
RoundToIntegerSP(SRC[31:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction := MXCSR:RC        ; get round control from MXCSR
    else
        rounding_direction := imm8[1:0]       ; get round control from imm8[1:0]
    FI
    M := imm8[7:4]            ; get the scaling factor

    case (rounding_direction)
    00: TMP[31:0] := round_to_nearest_even_integer(2^M*SRC[31:0])
    01: TMP[31:0] := round_to_equal_or_smaller_integer(2^M*SRC[31:0])
    10: TMP[31:0] := round_to_equal_or_larger_integer(2^M*SRC[31:0])
    11: TMP[31:0] := round_to_nearest_smallest_magnitude_integer(2^M*SRC[31:0])
    ESAC;

    Dest[31:0] := 2^-M* TMP[31:0]            ; scale down back to 2^-M
    if (imm8[3] = 0) Then          ; check SPE
        if (SRC[31:0] != Dest[31:0]) Then       ; check precision lost
            set_precision()            ; set #PE
        FI;
    FI;
    return(Dest[31:0])
}
```

VRNDSCALEPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF *src is a memory operand*
    THEN TMP_SRC := BROADCAST32(SRC, VL, k1)
    ELSE TMP_SRC := SRC
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := RoundToIntegerSP(TMP_SRC[i+31:i]), imm8[7:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+31:i] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRNDSCALEPS __m512 _mm512_roundscale_ps(__m512 a, int imm);
VRNDSCALEPS __m512 _mm512_roundscale_round_ps(__m512 a, int imm, int sae);
VRNDSCALEPS __m512 _mm512_mask_roundscale_ps(__m512 s, __mmask16 k, __m512 a, int imm);
VRNDSCALEPS __m512 _mm512_mask_roundscale_round_ps(__m512 s, __mmask16 k, __m512 a, int imm, int sae);
VRNDSCALEPS __m512 _mm512_maskz_roundscale_ps(__mmask16 k, __m512 a, int imm);
VRNDSCALEPS __m512 _mm512_maskz_roundscale_round_ps(__mmask16 k, __m512 a, int imm, int sae);
VRNDSCALEPS __m256 _mm256_roundscale_ps(__m256 a, int imm);
VRNDSCALEPS __m256 _mm256_mask_roundscale_ps(__m256 s, __mmask8 k, __m256 a, int imm);
VRNDSCALEPS __m256 _mm256_maskz_roundscale_ps(__mmask8 k, __m256 a, int imm);
VRNDSCALEPS __m128 _mm_roundscale_ps(__m256 a, int imm);
VRNDSCALEPS __m128 _mm_mask_roundscale_ps(__m128 s, __mmask8 k, __m128 a, int imm);
VRNDSCALEPS __m128 _mm_maskz_roundscale_ps(__mmask8 k, __m128 a, int imm);

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-48, "Type E2 Class Exception Conditions."

VRNDSCALESD—Round Scalar Float64 Value to Include a Given Number of Fraction Bits

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W1 0B /r ib VRNDSCALESD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	A	V/V	AVX512F OR AVX10.1	Rounds scalar double precision floating-point value in xmm3/m64 to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Rounds a double precision floating-point value in the low quadword (see Figure 5-29) element of the second source operand (the third operand) by the rounding mode specified in the immediate operand and places the result in the corresponding element of the destination operand (the first operand) according to the writemask. The quadword element at bits 127:64 of the destination is copied from the first source operand (the second operand).

The destination and first source operands are XMM registers, the 2nd source operand can be an XMM register or memory location. Bits MAXVL-1:128 of the destination register are cleared.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a double precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to `1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation for VRNDSCALESD is

ROUND(x) = 2^{-M}*Round_to_INT(x*2^{M}, round_ctrl),

round_ctrl = imm[3:0];

M=imm[7:4];

The operation of x*2^{M} is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

VRNDSCALESD is a more general form of the VEX-encoded VROUNDSD instruction. In VROUNDSD, the formula of the operation is

ROUND(x) = Round_to_INT(x, round_ctrl),

round_ctrl = imm[3:0];

EVEX encoded version: The source operand is a XMM register or a 64-bit memory location. The destination operand is a XMM register.

Handling of special case of input values are listed in Table 5-29.

Operation

```
RoundToIntegerDP(SRC[63:0], imm8[7:0]) {
```

```
    if (imm8[2] = 1)
        rounding_direction := MXCSR:RC        ; get round control from MXCSR
    else
        rounding_direction := imm8[1:0]       ; get round control from imm8[1:0]
    FI
    M := imm8[7:4]              ; get the scaling factor

    case (rounding_direction)
    00: TMP[63:0] := round_to_nearest_even_integer(2^M*SRC[63:0])
    01: TMP[63:0] := round_to_equal_or_smaller_integer(2^M*SRC[63:0])
    10: TMP[63:0] := round_to_equal_or_larger_integer(2^M*SRC[63:0])
    11: TMP[63:0] := round_to_nearest_smallest_magnitude_integer(2^M*SRC[63:0])
    ESAC

    Dest[63:0] := 2^-M* TMP[63:0]             ; scale down back to 2^-M

    if (imm8[3] = 0) Then    ; check SPE
        if (SRC[63:0] != Dest[63:0]) Then       ; check precision lost
            set_precision()                     ; set #PE
        FI;
    FI;
    return(Dest[63:0])
}
```

VRNDSCALESD (EVEX encoded version)

```
IF k1[0] or *no writemask*
    THEN      DEST[63:0] := RoundToIntegerDP(SRC2[63:0], Zero_upper_imm[7:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRNDSCALESD __m128d _mm_roundscale_sd (__m128d a, __m128d b, int imm);
VRNDSCALESD __m128d _mm_roundscale_round_sd (__m128d a, __m128d b, int imm, int sae);
VRNDSCALESD __m128d _mm_mask_roundscale_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d _mm_mask_roundscale_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm, int sae);
VRNDSCALESD __m128d _mm_maskz_roundscale_sd (__mmask8 k, __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d _mm_maskz_roundscale_round_sd (__mmask8 k, __m128d a, __m128d b, int imm, int sae);

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-49, "Type E3 Class Exception Conditions."

VRNDSCALESH—Round Scalar FP16 Value to Include a Given Number of Fraction Bits

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.NP.0F3A.W0 0A /r /ib VRNDSCALESH xmm1{k1}{z}, xmm2, xmm3/m16 {sae}, imm8	A	V/V	AVX512_FP16 OR AVX10.1	Round the low FP16 value in xmm3/m16 to a number of fraction bits specified by the imm8 field. Store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

This instruction rounds the low FP16 value in the second source operand by the rounding mode specified in the immediate operand (see Table 5-30) and places the result in the destination operand.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result), and returns the result as a FP16 value.

Note that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation. Three bit fields are defined and shown in Table 5-30, "Imm8 Controls for VRNDSCALEPH/VRNDSCALESH." Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control, and bits 1:0 specify a non-sticky rounding-mode value.

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN.

The sign of the result of this instruction is preserved, including the sign of zero. Special cases are described in Table 5-31.

If this instruction encoding's SPE bit (bit 3) in the immediate operand is 1, VRNDSCALESH can set MXCSR.UE without MXCSR.PE.

The formula of the operation on each data element for VRNDSCALESH is:

ROUND(x) = 2^{-M} *Round_to_INT(x * 2^{M}, round_ctrl),

round_ctrl = imm[3:0];

M=imm[7:4];

The operation of x * 2^{M} is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

Operation

VRNDSCALESH dest{k1}, src1, src2, imm8

```
IF k1[0] or *no writemask*:
    DEST.fp16[0] := round_fp16_to_integer(src2.fp16[0], imm8) // see VRNDSCALEPH
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
//else DEST.fp16[0] remains unchanged

DEST[127:16] = src1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRNDSCALESH __m128h _mm_mask_roundscale_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int imm8, const int sae);

VRNDSCALESH __m128h _mm_maskz_roundscale_round_sh (__mmask8 k, __m128h a, __m128h b, int imm8, const int sae);

VRNDSCALESH __m128h _mm_roundscale_round_sh (__m128h a, __m128h b, int imm8, const int sae);

VRNDSCALESH __m128h _mm_mask_roundscale_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int imm8);

VRNDSCALESH __m128h _mm_maskz_roundscale_sh (__mmask8 k, __m128h a, __m128h b, int imm8);

VRNDSCALESH __m128h _mm_roundscale_sh (__m128h a, __m128h b, int imm8);

SIMD Floating-Point Exceptions

Invalid, Underflow, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VRNDSCALESS—Round Scalar Float32 Value to Include a Given Number of Fraction Bits

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W0 0A /r ib VRNDSCALESS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512F OR AVX10.1	Rounds scalar single-precision floating-point value in xmm3/m32 to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Rounds the single precision floating-point value in the low doubleword element of the second source operand (the third operand) by the rounding mode specified in the immediate operand (see Figure 5-29) and places the result in the corresponding element of the destination operand (the first operand) according to the writemask. The double-word elements at bits 127:32 of the destination are copied from the first source operand (the second operand).

The destination and first source operands are XMM registers, the 2nd source operand can be an XMM register or memory location. Bits MAXVL-1:128 of the destination register are cleared.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a single precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (immediate control tables below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation for VRNDSCALESS is

ROUND(x) = 2^{-M}*Round_to_INT(x*2^{M}, round_ctrl),

round_ctrl = imm[3:0];

M=imm[7:4];

The operation of x*2^{M} is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

VRNDSCALESS is a more general form of the VEX-encoded VROUNDSS instruction. In VROUNDSS, the formula of the operation on each element is

ROUND(x) = Round_to_INT(x, round_ctrl),

round_ctrl = imm[3:0];

EVEX encoded version: The source operand is a XMM register or a 32-bit memory location. The destination operand is a XMM register.

Handling of special case of input values are listed in Table 5-29.

Operation

```
RoundToIntegerSP(SRC[31:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction := MXCSR:RC        ; get round control from MXCSR
    else
        rounding_direction := imm8[1:0]       ; get round control from imm8[1:0]
    FI
    M := imm8[7:4]            ; get the scaling factor

    case (rounding_direction)
    00: TMP[31:0] := round_to_nearest_even_integer(2^M*SRC[31:0])
    01: TMP[31:0] := round_to_equal_or_smaller_integer(2^M*SRC[31:0])
    10: TMP[31:0] := round_to_equal_or_larger_integer(2^M*SRC[31:0])
    11: TMP[31:0] := round_to_nearest_smallest_magnitude_integer(2^M*SRC[31:0])
    ESAC;

    Dest[31:0] := 2^-M* TMP[31:0]            ; scale down back to 2^-M
    if (imm8[3] = 0) Then          ; check SPE
        if (SRC[31:0] != Dest[31:0]) Then       ; check precision lost
            set_precision()             ; set #PE
        FI;
    FI;
    return(Dest[31:0])
}
```

VRNDSCALESS (EVEX encoded version)

```
IF k1[0] or *no writemask*
    THEN      DEST[31:0] := RoundToIntegerSP(SRC2[31:0], Zero_upper_imm[7:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VRNDSCALESS __m128 _mm_roundscale_ss ( __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_roundscale_round_ss ( __m128 a, __m128 b, int imm, int sae);
VRNDSCALESS __m128 _mm_mask_roundscale_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_mask_roundscale_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm, int sae);
VRNDSCALESS __m128 _mm_maskz_roundscale_ss ( __mmask8 k, __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_maskz_roundscale_round_ss ( __mmask8 k, __m128 a, __m128 b, int imm, int sae);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-49, "Type E3 Class Exception Conditions."

VRSQRT14PD—Compute Approximate Reciprocals of Square Roots of Packed Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 4E /r VRSQRT14PD xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Computes the approximate reciprocal square roots of the packed double precision floating-point values in xmm2/m128/m64bcst and stores the results in xmm1. Under writemask.
EVEX.256.66.0F38.W1 4E /r VRSQRT14PD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Computes the approximate reciprocal square roots of the packed double precision floating-point values in ymm2/m256/m64bcst and stores the results in ymm1. Under writemask.
EVEX.512.66.0F38.W1 4E /r VRSQRT14PD zmm1 {k1}{z}, zmm2/m512/m64bcst	A	V/V	AVX512F OR AVX10.1	Computes the approximate reciprocal square roots of the packed double precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1 under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocals of the square roots of the eight packed double precision floating-point values in the source operand (the second operand) and stores the packed double precision floating-point results in the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than 2^{-14}.

EVEX.512 encoded version: The source operand can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

The VRSQRT14PD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. When the source operand is an +∞ then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

A numerically exact implementation of VRSQRT14xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT14PD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+63:i] := APPROXIMATE(1.0/ SQRT(SRC[63:0]));
                ELSE DEST[i+63:i] := APPROXIMATE(1.0/ SQRT(SRC[i+63:i]));
            FI;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[i+63:i] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Table 5-32. VRSQRT14PD Special Cases

Input value	Result value	Comments
Any denormal	Normal	Cannot generate overflow
X = 2^{-2n}	2^n	
X < 0	QNaN_Indefinite	Including -INF
X = -0	-INF	
X = +0	+INF	
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT14PD __m512d _mm512_rsqrt14_pd(__m512d a);
VRSQRT14PD __m512d _mm512_mask_rsqrt14_pd(__m512d s, __mmask8 k, __m512d a);
VRSQRT14PD __m512d _mm512_maskz_rsqrt14_pd(__mmask8 k, __m512d a);
VRSQRT14PD __m256d _mm256_rsqrt14_pd(__m256d a);
VRSQRT14PD __m256d _mm512_mask_rsqrt14_pd(__m256d s, __mmask8 k, __m256d a);
VRSQRT14PD __m256d _mm512_maskz_rsqrt14_pd(__mmask8 k, __m256d a);
VRSQRT14PD __m128d _mm_rsqrt14_pd(__m128d a);
VRSQRT14PD __m128d _mm_mask_rsqrt14_pd(__m128d s, __mmask8 k, __m128d a);
VRSQRT14PD __m128d _mm_maskz_rsqrt14_pd(__mmask8 k, __m128d a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, "Type E4 Class Exception Conditions."

VRSQRT14PS—Compute Approximate Reciprocals of Square Roots of Packed Float32 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 4E /r VRSQRT14PS xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Computes the approximate reciprocal square roots of the packed single-precision floating-point values in xmm2/m128/m32bcst and stores the results in xmm1. Under writemask.
EVEX.256.66.0F38.W0 4E /r VRSQRT14PS ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Computes the approximate reciprocal square roots of the packed single-precision floating-point values in ymm2/m256/m32bcst and stores the results in ymm1. Under writemask.
EVEX.512.66.0F38.W0 4E /r VRSQRT14PS zmm1 {k1}{z}, zmm2/m512/m32bcst	A	V/V	AVX512F OR AVX10.1	Computes the approximate reciprocal square roots of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocals of the square roots of 16 packed single precision floating-point values in the source operand (the second operand) and stores the packed single precision floating-point results in the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than 2^{-14}.

EVEX.512 encoded version: The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

The VRSQRT14PS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. When the source operand is an +∞ then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

A numerically exact implementation of VRSQRT14xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT14PS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+31:i] := APPROXIMATE(1.0/ SQRT(SRC[31:0]));
                ELSE DEST[i+31:i] := APPROXIMATE(1.0/ SQRT(SRC[i+31:i]));
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+31:i] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Table 5-33. VRSQRT14PS Special Cases

Input value	Result value	Comments
Any denormal	Normal	Cannot generate overflow
X = 2^{-2n}	2^n	
X < 0	QNaN_Indefinite	Including -INF
X = -0	-INF	
X = +0	+INF	
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT14PS __m512 _mm512_rsqrt14_ps(__m512 a);
VRSQRT14PS __m512 _mm512_mask_rsqrt14_ps(__m512 s, __mmask16 k, __m512 a);
VRSQRT14PS __m512 _mm512_maskz_rsqrt14_ps(__mmask16 k, __m512 a);
VRSQRT14PS __m256 _mm256_rsqrt14_ps(__m256 a);
VRSQRT14PS __m256 _mm256_mask_rsqrt14_ps(__m256 s, __mmask8 k, __m256 a);
VRSQRT14PS __m256 _mm256_maskz_rsqrt14_ps(__mmask8 k, __m256 a);
VRSQRT14PS __m128 _mm_rsqrt14_ps(__m128 a);
VRSQRT14PS __m128 _mm_mask_rsqrt14_ps(__m128 s, __mmask8 k, __m128 a);
VRSQRT14PS __m128 _mm_maskz_rsqrt14_ps(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions."

VRSQRT14SD—Compute Approximate Reciprocal of Square Root of Scalar Float64 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W1 4F /r VRSQRT14SD xmm1 {k1}{z}, xmm2, xmm3/m64	A	V/V	AVX512F OR AVX10.1	Computes the approximate reciprocal square root of the scalar double precision floating-point value in xmm3/m64 and stores the result in the low quadword element of xmm1 using writemask k1. Bits[127:64] of xmm2 is copied to xmm1[127:64].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Computes the approximate reciprocal of the square roots of the scalar double precision floating-point value in the low quadword element of the source operand (the second operand) and stores the result in the low quadword element of the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than 2^{-14}. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

The VRSQRT14SD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. When the source operand is an +∞ then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRSQRT14xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT14SD (EVEX version)

```
IF k1[0] or *no writemask*
    THEN     DEST[63:0] := APPROXIMATE(1.0/ SQRT(SRC2[63:0]))
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Table 5-34. VRSQRT14SD Special Cases

Input value	Result value	Comments
Any denormal	Normal	Cannot generate overflow
X = 2^{-2n}	2^n	
X < 0	QNaN_Indefinite	Including -INF
X = -0	-INF	
X = +0	+INF	
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT14SD __m128d _mm_rsqrt14_sd(__m128d a, __m128d b);
VRSQRT14SD __m128d _mm_mask_rsqrt14_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VRSQRT14SD __m128d _mm_maskz_rsqrt14_sd(__mmask8d m, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-60, "Type E10 Class Exception Conditions."

VRSQRT14SS—Compute Approximate Reciprocal of Square Root of Scalar Float32 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W0 4F /r VRSQRT14SS xmm1 {k1}{z}, xmm2, xmm3/m32	A	V/V	AVX512F OR AVX10.1	Computes the approximate reciprocal square root of the scalar single-precision floating-point value in xmm3/m32 and stores the result in the low doubleword element of xmm1 using writemask k1. Bits[127:32] of xmm2 is copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Computes of the approximate reciprocal of the square root of the scalar single precision floating-point value in the low doubleword element of the source operand (the second operand) and stores the result in the low doubleword element of the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than 2^{-14}. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

The VRSQRT14SS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. When the source operand is an ∞, zero with the sign of the source value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRSQRT14xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT14SS (EVEX version)

```
IF k1[0] or *no writemask*
    THEN     DEST[31:0] := APPROXIMATE(1.0/ SQRT(SRC2[31:0]))
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Table 5-35. VRSQRT14SS Special Cases

Input value	Result value	Comments
Any denormal	Normal	Cannot generate overflow
$X = 2^{-2n}$	2^n	
X < 0	QNaN_Indefinite	Including -INF
X = -0	-INF	
X = +0	+INF	
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT14SS __m128 _mm_rsqrt14_ss(__m128 a, __m128 b);
VRSQRT14SS __m128 _mm_mask_rsqrt14_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VRSQRT14SS __m128 _mm_maskz_rsqrt14_ss(__mmask8 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-60, "Type E10 Class Exception Conditions."

VRSQRTPH—Compute Reciprocals of Square Roots of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP6.W0 4E /r VRSQRTPH xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Compute the approximate reciprocals of the square roots of packed FP16 values in xmm2/m128/m16bcst and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 4E /r VRSQRTPH ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Compute the approximate reciprocals of the square roots of packed FP16 values in ymm2/m256/m16bcst and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 4E /r VRSQRTPH zmm1{k1}{z}, zmm2/m512/m16bcst	A	V/V	AVX512_FP16 OR AVX10.1	Compute the approximate reciprocals of the square roots of packed FP16 values in zmm2/m512/m16bcst and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocals square-root of 8/16/32 packed FP16 floating-point values in the source operand (the second operand) and stores the packed FP16 floating-point results in the destination operand.

The maximum relative error for this approximation is less than $2^{-11} + 2^{-14}$. For special cases, see Table 5-36.

The destination elements are updated according to the writemask.

Table 5-36. VRSQRTPH/VRSQRTSH Special Cases

Input value	Reset Value	Comments
Any denormal	Normal	Cannot generate overflow
$X = 2^{-2n}$	2^n	
$X < 0$	QNaN_Indefinite	Including $-\infty$
$X = -0$	$-\infty$	
$X = +0$	$+\infty$	
$X = +\infty$	$+0$	

Operation

VRSQRTPH dest{k1}, src

```
VL = 128, 256 or 512
KL := VL/16

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.fp16[0]
        ELSE:
            tsrc := src.fp16[i]
        DEST.fp16[i] := APPROXIMATE(1.0 / SQRT(tsrc) )
    ELSE IF *zeroing*:
        DEST.fp16[i] := 0
    //else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRTPH __m128h _mm_mask_rsqrt_ph (__m128h src, __mmask8 k, __m128h a);
VRSQRTPH __m128h _mm_maskz_rsqrt_ph (__mmask8 k, __m128h a);
VRSQRTPH __m128h _mm_rsqrt_ph (__m128h a);
VRSQRTPH __m256h _mm256_mask_rsqrt_ph (__m256h src, __mmask16 k, __m256h a);
VRSQRTPH __m256h _mm256_maskz_rsqrt_ph (__mmask16 k, __m256h a);
VRSQRTPH __m256h _mm256_rsqrt_ph (__m256h a);
VRSQRTPH __m512h _mm512_mask_rsqrt_ph (__m512h src, __mmask32 k, __m512h a);
VRSQRTPH __m512h _mm512_maskz_rsqrt_ph (__mmask32 k, __m512h a);
VRSQRTPH __m512h _mm512_rsqrt_ph (__m512h a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

VRSQRTSH—Compute Approximate Reciprocal of Square Root of Scalar FP16 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.MAP6.W0 4F /r VRSQRTSH xmm1{k1}{z}, xmm2, xmm3/m16	A	V/V	AVX512_FP16 OR AVX10.1	Compute the approximate reciprocal square root of the FP16 value in xmm3/m16 and store the result in the low word element of xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs the computation of the approximate reciprocal square-root of the low FP16 value in the second source operand (the third operand) and stores the result in the low word element of the destination operand (the first operand) according to the writemask k1.

The maximum relative error for this approximation is less than $2^{-11} + 2^{-14}$.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL−1:128 of the destination operand are zeroed.

For special cases, see Table 5-36.

Operation

VRSQRTSH dest{k1}, src1, src2

```
VL = 128, 256 or 512
KL := VL/16

IF k1[0] or *no writemask*:
    DEST.fp16[0] := APPROXIMATE(1.0 / SQRT(src2.fp16[0]))
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
//else DEST.fp16[0] remains unchanged
DEST[127:16] := src1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRTSH __m128h _mm_mask_rsqrt_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VRSQRTSH __m128h _mm_maskz_rsqrt_sh (__mmask8 k, __m128h a, __m128h b);
VRSQRTSH __m128h _mm_rsqrt_sh (__m128h a, __m128h b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-60, “Type E10 Class Exception Conditions.”

VSCALEFPD—Scale Packed Float64 Values With Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 2C /r VSCALEFPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Scale the packed double precision floating-point values in xmm2 using values from xmm3/m128/m64bcst. Under writemask k1.
EVEX.256.66.0F38.W1 2C /r VSCALEFPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Scale the packed double precision floating-point values in ymm2 using values from ymm3/m256/m64bcst. Under writemask k1.
EVEX.512.66.0F38.W1 2C /r VSCALEFPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	A	V/V	AVX512F OR AVX10.1	Scale the packed double precision floating-point values in zmm2 using values from zmm3/m512/m64bcst. Under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a floating-point scale of the packed double precision floating-point values in the first source operand by multiplying them by 2 to the power of the double precision floating-point values in second source operand.

The equation of this operation is given by:

$zmm1 := zmm2 * 2^{floor(zmm3)}$.

Floor(zmm3) means maximum integer value ≤ zmm3.

If the result cannot be represented in double precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-37 and Table 5-38.

Table 5-37. VSCALEFPD/SD/PS/SS Special Cases

		Src2				Set IE
		±NaN	**+Inf**	**-Inf**	**0/Denorm/Norm**	
Src1	±QNaN	QNaN(Src1)	+INF	+0	QNaN(Src1)	IF either source is SNAN
	±SNaN	QNaN(Src1)	QNaN(Src1)	QNaN(Src1)	QNaN(Src1)	YES
	±Inf	QNaN(Src2)	Src1	QNaN_Indefinite	Src1	IF Src2 is SNAN or -INF
	±0	QNaN(Src2)	QNaN_Indefinite	Src1	Src1	IF Src2 is SNAN or +INF
	Denorm/Norm	QNaN(Src2)	±INF (Src1 sign)	±0 (Src1 sign)	Compute Result	IF Src2 is SNAN

Table 5-38. Additional VSCALEFPD/SD Special Cases

Special Case	Returned value	Faults
$\|result\| < 2^{-1074}$	±0 or ±Min-Denormal (Src1 sign)	Underflow
$\|result\| \geq 2^{1024}$	±INF (Src1 sign) or ±Max-normal (Src1 sign)	Overflow

Operation

```
SCALE(SRC1, SRC2)
{
TMP_SRC2 := SRC2
TMP_SRC1 := SRC1
IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
/* SRC2 is a 64 bits floating-point value */
DEST[63:0] := TMP_SRC1[63:0] * POW(2, Floor(TMP_SRC2[63:0]))
}
VSCALEFPD (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := SCALE(SRC1[i+63:i], SRC2[63:0]);
                ELSE DEST[i+63:i] := SCALE(SRC1[i+63:i], SRC2[i+63:i]);
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VSCALEFPD __m512d _mm512_scalef_round_pd(__m512d a, __m512d b, int rounding);
VSCALEFPD __m512d _mm512_mask_scalef_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int rounding);
VSCALEFPD __m512d _mm512_maskz_scalef_round_pd(__mmask8 k, __m512d a, __m512d b, int rounding);
VSCALEFPD __m512d _mm512_scalef_pd(__m512d a, __m512d b);
VSCALEFPD __m512d _mm512_mask_scalef_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VSCALEFPD __m512d _mm512_maskz_scalef_pd(__mmask8 k, __m512d a, __m512d b);
VSCALEFPD __m256d _mm256_scalef_pd(__m256d a, __m256d b);
VSCALEFPD __m256d _mm256_mask_scalef_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VSCALEFPD __m256d _mm256_maskz_scalef_pd(__mmask8 k, __m256d a, __m256d b);
VSCALEFPD __m128d _mm_scalef_pd(__m128d a, __m128d b);
VSCALEFPD __m128d _mm_mask_scalef_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VSCALEFPD __m128d _mm_maskz_scalef_pd(__mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions

See Table 2-48, "Type E2 Class Exception Conditions."

VSCALEFPH—Scale Packed FP16 Values with FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP6.W0 2C /r VSCALEFPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Scale the packed FP16 values in xmm2 using values from xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 2C /r VSCALEFPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Scale the packed FP16 values in ymm2 using values from ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 2C /r VSCALEFPH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Scale the packed FP16 values in zmm2 using values from zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a floating-point scale of the packed FP16 values in the first source operand by multiplying it by 2 to the power of the FP16 values in second source operand. The destination elements are updated according to the writemask.

The equation of this operation is given by:

zmm1 := zmm2 * $2^{floor(zmm3)}$.

Floor(zmm3) means maximum integer value ≤ zmm3.

If the result cannot be represented in FP16, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand), is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits), and on the SAE bit.

Handling of special-case input values are listed in Table 5-39 and Table 5-40.

Table 5-39. VSCALEFPH/VSCALEFSH Special Cases

Src1	Src2				Set IE
	±NaN	+INF	−INF	0/Denorm/Norm	
±QNaN	QNaN(Src1)	+INF	+0	QNaN(Src1)	IF either source is SNaN
±SNaN	QNaN(Src1)	QNaN(Src1)	QNaN(Src1)	QNaN(Src1)	YES
±INF	QNaN(Src2)	Src1	QNaN_Indefinite	Src1	IF Src2 is SNaN or −INF
±0	QNaN(Src2)	QNaN_Indefinite	Src1	Src1	IF Src2 is SNaN or +INF
Denorm/Norm	QNaN(Src2)	±INF (Src1 sign)	±0 (Src1 sign)	Compute Result	IF Src2 is SNaN

Table 5-40. Additional VSCALEFPH/VSCALEFSH Special Cases

Special Case	Returned Value	Faults
|result| < 2^{-24}	±0 or ±Min-Denormal (Src1 sign)	Underflow
|result| ≥ 2^{16}	±INF (Src1 sign) or ±Max-Denormal (Src1 sign)	Overflow

Operation

```
def scale_fp16(src1,src2):
    tmp1 := src1
    tmp2 := src2
    return tmp1 * POW(2, FLOOR(tmp2))

VSCALEFPH dest{k1}, src1, src2
VL = 128, 256, or 512
KL := VL / 16

IF (VL = 512) AND (EVEX.b = 1) and no memory operand:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC2 is memory and (EVEX.b = 1):
            tsrc := src2.fp16[0]
        ELSE:
            tsrc := src2.fp16[i]
        dest.fp16[i] := scale_fp16(src1.fp16[i],tsrc)
    ELSE IF *zeroing*:
        dest.fp16[i] := 0
    //else dest.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VSCALEFPH __m128h _mm_mask_scalef_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VSCALEFPH __m128h _mm_maskz_scalef_ph (__mmask8 k, __m128h a, __m128h b);
VSCALEFPH __m128h _mm_scalef_ph (__m128h a, __m128h b);
VSCALEFPH __m256h _mm256_mask_scalef_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VSCALEFPH __m256h _mm256_maskz_scalef_ph (__mmask16 k, __m256h a, __m256h b);
VSCALEFPH __m256h _mm256_scalef_ph (__m256h a, __m256h b);
VSCALEFPH __m512h _mm512_mask_scalef_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VSCALEFPH __m512h _mm512_maskz_scalef_ph (__mmask32 k, __m512h a, __m512h b);
VSCALEFPH __m512h _mm512_scalef_ph (__m512h a, __m512h b);
VSCALEFPH __m512h _mm512_mask_scalef_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, const int rounding);
VSCALEFPH __m512h _mm512_maskz_scalef_round_ph (__mmask32 k, __m512h a, __m512h b, const int;
VSCALEFPH __m512h _mm512_scalef_round_ph (__m512h a, __m512h b, const int rounding);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).

Denormal is not reported for Src2.

Other Exceptions

EVEX-encoded instruction, see Table 2-48, “Type E2 Class Exception Conditions”.

VSCALEFPS—Scale Packed Float32 Values With Float32 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 2C /r VSCALEFPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Scale the packed single-precision floating-point values in xmm2 using values from xmm3/m128/m32bcst. Under writemask k1.
EVEX.256.66.0F38.W0 2C /r VSCALEFPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Scale the packed single-precision values in ymm2 using floating-point values from ymm3/m256/m32bcst. Under writemask k1.
EVEX.512.66.0F38.W0 2C /r VSCALEFPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	A	V/V	AVX512F OR AVX10.1	Scale the packed single-precision floating-point values in zmm2 using floating-point values from zmm3/m512/m32bcst. Under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a floating-point scale of the packed single precision floating-point values in the first source operand by multiplying them by 2 to the power of the float32 values in second source operand.

The equation of this operation is given by:

zmm1 := zmm2*$2^{floor(zmm3)}$.

Floor(zmm3) means maximum integer value ≤ zmm3.

If the result cannot be represented in single precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Handling of special-case input values are listed in Table 5-37 and Table 5-41.

Table 5-41. Additional VSCALEFPS/SS Special Cases

Special Case	Returned value	Faults
|result| < 2^{-149}	±0 or ±Min-Denormal (Src1 sign)	Underflow
|result| ≥ 2^{128}	±INF (Src1 sign) or ±Max-normal (Src1 sign)	Overflow

Operation

```
SCALE(SRC1, SRC2)
{               ; Check for denormal operands
TMP_SRC2 := SRC2
TMP_SRC1 := SRC1
IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
/* SRC2 is a 32 bits floating-point value */
DEST[31:0] := TMP_SRC1[31:0] * POW(2, Floor(TMP_SRC2[31:0]))
}
```

VSCALEFPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := SCALE(SRC1[i+31:i], SRC2[31:0]);
                ELSE DEST[i+31:i] := SCALE(SRC1[i+31:i], SRC2[i+31:i]);
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSCALEFPS __m512 _mm512_scalef_round_ps(__m512 a, __m512 b, int rounding);
VSCALEFPS __m512 _mm512_mask_scalef_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int rounding);
VSCALEFPS __m512 _mm512_maskz_scalef_round_ps(__mmask16 k, __m512 a, __m512 b, int rounding);
VSCALEFPS __m512 _mm512_scalef_ps(__m512 a, __m512 b);
VSCALEFPS __m512 _mm512_mask_scalef_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VSCALEFPS __m512 _mm512_maskz_scalef_ps(__mmask16 k, __m512 a, __m512 b);
VSCALEFPS __m256 _mm256_scalef_ps(__m256 a, __m256 b);
VSCALEFPS __m256 _mm256_mask_scalef_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VSCALEFPS __m256 _mm256_maskz_scalef_ps(__mmask8 k, __m256 a, __m256 b);
VSCALEFPS __m128 _mm_scalef_ps(__m128 a, __m128 b);
VSCALEFPS __m128 _mm_mask_scalef_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VSCALEFPS __m128 _mm_maskz_scalef_ps(__mmask8 k, __m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions

See Table 2-48, "Type E2 Class Exception Conditions."

VSCALEFSD—Scale Scalar Float64 Values With Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W1 2D /r VSCALEFSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	A	V/V	AVX512F OR AVX10.1	Scale the scalar double precision floating-point values in xmm2 using the value from xmm3/m64. Under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a floating-point scale of the scalar double precision floating-point value in the first source operand by multiplying it by 2 to the power of the double precision floating-point value in second source operand.

The equation of this operation is given by:

$xmm1 := xmm2 * 2^{floor(xmm3)}$.

Floor(xmm3) means maximum integer value ≤ xmm3.

If the result cannot be represented in double precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX encoded version: The first source operand is an XMM register. The second source operand is an XMM register or a memory location. The destination operand is an XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-37 and Table 5-38.

Operation

```
SCALE(SRC1, SRC2)
{
    ; Check for denormal operands
TMP_SRC2 := SRC2
TMP_SRC1 := SRC1
IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
/* SRC2 is a 64 bits floating-point value */
DEST[63:0] := TMP_SRC1[63:0] * POW(2, Floor(TMP_SRC2[63:0]))
}
```

VSCALEFSD (EVEX encoded version)

```
IF (EVEX.b= 1) and SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] OR *no writemask*
    THEN DEST[63:0] := SCALE(SRC1[63:0], SRC2[63:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63:0] := 0
        FI
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VSCALEFSD __m128d _mm_scalef_round_sd(__m128d a, __m128d b, int);
VSCALEFSD __m128d _mm_mask_scalef_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSCALEFSD __m128d _mm_maskz_scalef_round_sd(__mmask8 k, __m128d a, __m128d b, int);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions

See Table 2-49, "Type E3 Class Exception Conditions."

VSCALEFSH—Scale Scalar FP16 Values with FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.MAP6.W0 2D /r VSCALEFSH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512_FP16 OR AVX10.1	Scale the FP16 values in xmm2 using the value from xmm3/m16 and store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a floating-point scale of the low FP16 element in the first source operand by multiplying it by 2 to the power of the low FP16 element in second source operand, storing the result in the low element of the destination operand.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

The equation of this operation is given by:

$$xmm1 := xmm2 * 2^{floor(xmm3)}.$$

Floor(xmm3) means maximum integer value ≤ xmm3.

If the result cannot be represented in FP16, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand), is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

Handling of special-case input values are listed in Table 5-39 and Table 5-40.

Operation

VSCALEFSH dest{k1}, src1, src2

```
IF (EVEX.b = 1) and no memory operand:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] or *no writemask*:
    dest.fp16[0] := scale_fp16(src1.fp16[0], src2.fp16[0]) // see VSCALEFPH
ELSE IF *zeroing*:
    dest.fp16[0] := 0
//else DEST.fp16[0] remains unchanged

DEST[127:16] := src1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VSCALEFSH __m128h _mm_mask_scalef_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, const int rounding);
VSCALEFSH __m128h _mm_maskz_scalef_round_sh (__mmask8 k, __m128h a, __m128h b, const int rounding);
VSCALEFSH __m128h _mm_scalef_round_sh (__m128h a, __m128h b, const int rounding);
VSCALEFSH __m128h _mm_mask_scalef_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VSCALEFSH __m128h _mm_maskz_scalef_sh (__mmask8 k, __m128h a, __m128h b);
VSCALEFSH __m128h _mm_scalef_sh (__m128h a, __m128h b);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-49, "Type E3 Class Exception Conditions."

Denormal-operand exception (#D) is checked and signaled for src1 operand, but not for src2 operand. The denormal-operand exception is checked for src1 operand only if the src2 operand is not NaN. If the src2 operand is NaN, the processor generates NaN and does not signal denormal-operand exception, even if src1 operand is denormal.

VSCALEFSS—Scale Scalar Float32 Value With Float32 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W0 2D /r VSCALEFSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	A	V/V	AVX512F OR AVX10.1	Scale the scalar single-precision floating-point value in xmm2 using floating-point value from xmm3/m32. Under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a floating-point scale of the scalar single precision floating-point value in the first source operand by multiplying it by 2 to the power of the float32 value in second source operand.

The equation of this operation is given by:

$xmm1 := xmm2*2^{floor(xmm3)}$.

Floor(xmm3) means maximum integer value ≤ xmm3.

If the result cannot be represented in single precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX encoded version: The first source operand is an XMM register. The second source operand is an XMM register or a memory location. The destination operand is an XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-37 and Table 5-41.

Operation

```
SCALE(SRC1, SRC2)
{
                ; Check for denormal operands
TMP_SRC2 := SRC2
TMP_SRC1 := SRC1
IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
/* SRC2 is a 32 bits floating-point value */
DEST[31:0] := TMP_SRC1[31:0] * POW(2, Floor(TMP_SRC2[31:0]))
}
```

VSCALEFSS (EVEX encoded version)

```
IF (EVEX.b= 1) and SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] OR *no writemask*
    THEN DEST[31:0] := SCALE(SRC1[31:0], SRC2[31:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] := 0
        FI
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VSCALEFSS __m128 _mm_scalef_round_ss(__m128 a, __m128 b, int);
VSCALEFSS __m128 _mm_mask_scalef_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSCALEFSS __m128 _mm_maskz_scalef_round_ss(__mmask8 k, __m128 a, __m128 b, int);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions

See Table 2-49, "Type E3 Class Exception Conditions."

VSCATTERDPS/VSCATTERDPD/VSCATTERQPS/VSCATTERQPD—Scatter Packed Single Precision, Packed Double Precision Floating-Point Values with Signed Dword and Qword Indices

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 A2 /vsib VSCATTERDPS vm32x {k1}, xmm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W0 A2 /vsib VSCATTERDPS vm32y {k1}, ymm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W0 A2 /vsib VSCATTERDPS vm32z {k1}, zmm1	A	V/V	AVX512F OR AVX10.1	Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.128.66.0F38.W1 A2 /vsib VSCATTERDPD vm32x {k1}, xmm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, scatter double precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W1 A2 /vsib VSCATTERDPD vm32y {k1}, ymm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, scatter double precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W1 A2 /vsib VSCATTERDPD vm32z {k1}, zmm1	A	V/V	AVX512F OR AVX10.1	Using signed dword indices, scatter double precision floating-point values to memory using writemask k1.
EVEX.128.66.0F38.W0 A3 /vsib VSCATTERQPS vm64x {k1}, xmm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W0 A3 /vsib VSCATTERQPS vm64y {k1}, xmm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W0 A3 /vsib VSCATTERQPS vm64z {k1}, ymm1	A	V/V	AVX512F OR AVX10.1	Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.128.66.0F38.W1 A3 /vsib VSCATTERQPD vm64x {k1}, xmm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, scatter double precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W1 A3 /vsib VSCATTERQPD vm64y {k1}, ymm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, scatter double precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W1 A3 /vsib VSCATTERQPD vm64z {k1}, zmm1	A	V/V	AVX512F OR AVX10.1	Using signed qword indices, scatter double precision floating-point values to memory using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	ModRM:reg (r)	N/A	N/A

Description

Stores up to four, eight, or 16 single precision elements (or two, four, or eight double precision elements) in double-word/quadword vector xmm1, ymm1, or zmm1, to the memory locations pointed by base address BASE_ADDR and index vector VINDEX, with scale SCALE. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be stored if their corresponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already scattered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated. If any traps or interrupts are pending from already scattered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

Note that:

- Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Note that this also include partially overlapping vector indices. Writes that are not overlapped may happen in any order. Memory ordering with other instructions follows the Intel-64 memory ordering model. Note that this does not account for non-overlapping indices that map into the same physical address locations.
- If two or more destination indices completely overlap, the "earlier" write(s) may be skipped.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the source register xmm, ymm, or zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be scattered in any order, but faults must be delivered in a right-to left order; thus, elements to the left of a faulting one may be scattered 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 scattered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- If this instruction overwrites itself and then takes a fault, only a subset of elements may be completed before the fault is delivered (as described above). If the fault handler completes and attempts to re-execute this instruction, the new instruction will be executed, and the scatter will not complete.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the k0 mask register is specified.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1 or 4 byte displacement

VSCATTERDPS (EVEX encoded versions)

```
(KL, VL)= (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR +SignExtend(VINDEX[i+31:i]) * SCALE + DISP] :=
            SRC[i+31:i]
            k1[j] := 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
```

VSCATTERDPD (EVEX encoded versions)

```
(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR +SignExtend(VINDEX[k+31:k]) * SCALE + DISP] :=
            SRC[i+63:i]
            k1[j] := 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
```

VSCATTERQPS (EVEX encoded versions)

```
(KL, VL)= (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP] :=
            SRC[i+31:i]
            k1[j] := 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
```

VSCATTERQPD (EVEX encoded versions)

```
(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + (VINDEX[i+63:i]) * SCALE + DISP] :=
            SRC[i+63:i]
            k1[j] := 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VSCATTERDPD void _mm512_i32scatter_pd(void * base, __m512i vdx, __m512d a, int scale);
VSCATTERDPD void _mm512_mask_i32scatter_pd(void * base, __mmask8 k, __m512i vdx, __m512d a, int scale);
VSCATTERDPS void _mm512_i32scatter_ps(void * base, __m512i vdx, __m512 a, int scale);
VSCATTERDPS void _mm512_mask_i32scatter_ps(void * base, __mmask16 k, __m512i vdx, __m512 a, int scale);
VSCATTERQPD void _mm512_i64scatter_pd(void * base, __m512i vdx, __m512d a, int scale);
VSCATTERQPD void _mm512_mask_i64scatter_pd(void * base, __mmask8 k, __m512i vdx, __m512d a, int scale);
VSCATTERQPS void _mm512_i64scatter_ps(void * base, __m512i vdx, __m512 a, int scale);
VSCATTERQPS void _mm512_mask_i64scatter_ps(void * base, __mmask8 k, __m512i vdx, __m512 a, int scale);
VSCATTERDPD void _mm256_i32scatter_pd(void * base, __m256i vdx, __m256d a, int scale);
VSCATTERDPD void _mm256_mask_i32scatter_pd(void * base, __mmask8 k, __m256i vdx, __m256d a, int scale);
VSCATTERDPS void _mm256_i32scatter_ps(void * base, __m256i vdx, __m256 a, int scale);
VSCATTERDPS void _mm256_mask_i32scatter_ps(void * base, __mmask8 k, __m256i vdx, __m256 a, int scale);
VSCATTERQPD void _mm256_i64scatter_pd(void * base, __m256i vdx, __m256d a, int scale);
VSCATTERQPD void _mm256_mask_i64scatter_pd(void * base, __mmask8 k, __m256i vdx, __m256d a, int scale);
VSCATTERQPS void _mm256_i64scatter_ps(void * base, __m256i vdx, __m256 a, int scale);
VSCATTERQPS void _mm256_mask_i64scatter_ps(void * base, __mmask8 k, __m256i vdx, __m256 a, int scale);
VSCATTERDPD void _mm_i32scatter_pd(void * base, __m128i vdx, __m128d a, int scale);
VSCATTERDPD void _mm_mask_i32scatter_pd(void * base, __mmask8 k, __m128i vdx, __m128d a, int scale);
VSCATTERDPS void _mm_i32scatter_ps(void * base, __m128i vdx, __m128 a, int scale);
VSCATTERDPS void _mm_mask_i32scatter_ps(void * base, __mmask8 k, __m128i vdx, __m128 a, int scale);
VSCATTERQPD void _mm_i64scatter_pd(void * base, __m128i vdx, __m128d a, int scale);
VSCATTERQPD void _mm_mask_i64scatter_pd(void * base, __mmask8 k, __m128i vdx, __m128d a, int scale);
VSCATTERQPS void _mm_i64scatter_ps(void * base, __m128i vdx, __m128 a, int scale);
VSCATTERQPS void _mm_mask_i64scatter_ps(void * base, __mmask8 k, __m128i vdx, __m128 a, int scale);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, "Type E12 Class Exception Conditions."

VSHA512MSG1—Perform an Intermediate Calculation for the Next Four SHA512 Message Qwords

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.F2.0F38.W0 CC 11:rrr:bbb VSHA512MSG1 ymm1, xmm2	A	V/V	AVX SHA512	Performs an intermediate calculation for the next four SHA512 message qwords using previous message qwords from ymm1 and xmm2, storing the result in ymm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A

Description

The VSHA512MSG1 instruction is one of the two SHA512 message scheduling instructions. The instruction performs an intermediate calculation for the next four SHA512 message qwords.

See https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf for more information on the SHA512 standard.

Operation

```
define ROR64(qword, n):
    count := n % 64
    dest := (qword >> count) | (qword << (64-count))
    return dest

define SHR64(qword, n):
    return qword >> n

define s0(qword):
    return ROR64(qword,1) ^ ROR64(qword, 8) ^ SHR64(qword, 7)

VSHA512MSG1 SRCDEST, SRC1
W[4] := SRC1.qword[0]
W[3] := SRCDEST.qword[3]
W[2] := SRCDEST.qword[2]
W[1] := SRCDEST.qword[1]
W[0] := SRCDEST.qword[0]

SRCDEST.qword[3] := W[3] + s0(W[4])
SRCDEST.qword[2] := W[2] + s0(W[3])
SRCDEST.qword[1] := W[1] + s0(W[2])
SRCDEST.qword[0] := W[0] + s0(W[1])
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VSHA512MSG1 __m256i _mm256_sha512msg1_epi64 (__m256i __A, __m128i __B);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-23, "Type 6 Class Exception Conditions."

VSHA512MSG2—Perform a Final Calculation for the Next Four SHA512 Message Qwords

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.F2.0F38.W0 CD 11:rrr:bbb VSHA512MSG2 ymm1, ymm2	A	V/V	AVX SHA512	Performs the final calculation for the next four SHA512 message qwords using previous message qwords from ymm1 and ymm2, storing the result in ymm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A

Description

The VSHA512MSG2 instruction is one of the two SHA512 message scheduling instructions. The instruction performs the final calculation for the next four SHA512 message qwords.

See https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf for more information on the SHA512 standard.

Operation

```
define ROR64(qword, n):
    count := n % 64
    dest := (qword >> count) | (qword << (64-count))
    return dest

define SHR64(qword, n):
    return qword >> n

define s1(qword):
    return ROR64(qword,19) ^ ROR64(qword, 61) ^ SHR64(qword, 6)
```

VSHA512MSG2 SRCDEST, SRC1

```
W[14] := SRC1.qword[2]
W[15] := SRC1.qword[3]
W[16] := SRCDEST.qword[0] + s1(W[14])
W[17] := SRCDEST.qword[1] + s1(W[15])
W[18] := SRCDEST.qword[2] + s1(W[16])
W[19] := SRCDEST.qword[3] + s1(W[17])

SRCDEST.qword[3] := W[19]
SRCDEST.qword[2] := W[18]
SRCDEST.qword[1] := W[17]
SRCDEST.qword[0] := W[16]
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VSHA512MSG2 __m256i _mm256_sha512msg2_epi64 (__m256i __A, __m256i __B);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-23, "Type 6 Class Exception Conditions."

VSHA512RNDS2—Perform Two Rounds of SHA512 Operation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.F2.0F38.W0 CB 11:rrr:bbb VSHA512RNDS2 ymm1, ymm2, xmm3	A	V/V	AVX SHA512	Perform 2 rounds of SHA512 operation using an initial SHA512 state (C,D,G,H) from ymm1, an initial SHA512 state (A,B,E,F) from ymm2, and a pre-computed sum of the next 2 round message qwords and the corresponding round constants from xmm3, storing the updated SHA512 state (A,B,E,F) result in ymm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

The VSHA512RNDS2 instruction performs two rounds of SHA512 operation using initial SHA512 state (C,D,G,H) from the first operand, an initial SHA512 state (A,B,E,F) from the second operand, and a pre-computed sum of the next two round message qwords and the corresponding round constants from the third operand (only the two lower qwords of the third operand). The updated SHA512 state (A,B,E,F) is written to the first operand, and the second operand can be used as the updated state (C,D,G,H) in later rounds.

See https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf for more information on the SHA512 standard.

Operation

```
define ROR64(qword, n):
    count := n % 64
    dest := (qword >> count) | (qword << (64-count))
    return dest

define SHR64(qword, n):
    return qword >> n

define cap_sigma0(qword):
    return ROR64(qword,28) ^ ROR64(qword, 34) ^ ROR64(qword, 39)

define cap_sigma1(qword):
    return ROR64(qword,14) ^ ROR64(qword, 18) ^ ROR64(qword, 41)

define MAJ(a,b,c):
    return (a & b) ^ (a & c) ^ (b & c)

define CH(e,f,g):
    return (e & f) ^ (g & ~e)
```

VSHA512RNDS2 SRCDEST, SRC1, SRC2

```
A[0] := SRC1.qword[3]
B[0] := SRC1.qword[2]
C[0] := SRCDEST.qword[3]
D[0] := SRCDEST.qword[2]
E[0] := SRC1.qword[1]
F[0] := SRC1.qword[0]
G[0] := SRCDEST.qword[1]
H[0] := SRCDEST.qword[0]
WK[0]:= SRC2.qword[0]
WK[1]:= SRC2.qword[1]

FOR i in 0..1:
    A[i+1] := CH(E[i], F[i], G[i]) +
          cap_sigma1(E[i]) + WK[i] + H[i] +
          MAJ(A[i], B[i], C[i]) +
          cap_sigma0(A[i])
    B[i+1] := A[i]
    C[i+1] := B[i]
    D[i+1] := C[i]
    E[i+1] := CH(E[i], F[i], G[i]) +
          cap_sigma1(E[i]) + WK[i] + H[i] + D[i]
    F[i+1] := E[i]
    G[i+1] := F[i]
    H[i+1] := G[i]

SRCDEST.qword[3] = A[2]
SRCDEST.qword[2] = B[2]
SRCDEST.qword[1] = E[2]
SRCDEST.qword[0] = F[2]
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VSHA512RNDS2 __m256i _mm256_sha512rnds2_epi64 (__m256i __A, __m256i __B, __m128i __C);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-23, “Type 6 Class Exception Conditions.”

VSHUFF32x4/VSHUFF64x2/VSHUFI32x4/VSHUFI64x2—Shuffle Packed Values at 128-Bit Granularity

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F3A.W0 23 /r ib VSHUFF32X4 ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shuffle 128-bit packed single-precision floating-point values selected by imm8 from ymm2 and ymm3/m256/m32bcst and place results in ymm1 subject to writemask k1.
EVEX.512.66.0F3A.W0 23 /r ib VSHUFF32x4 zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	A	V/V	AVX512F OR AVX10.1	Shuffle 128-bit packed single-precision floating-point values selected by imm8 from zmm2 and zmm3/m512/m32bcst and place results in zmm1 subject to writemask k1.
EVEX.256.66.0F3A.W1 23 /r ib VSHUFF64X2 ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shuffle 128-bit packed double precision floating-point values selected by imm8 from ymm2 and ymm3/m256/m64bcst and place results in ymm1 subject to writemask k1.
EVEX.512.66.0F3A.W1 23 /r ib VSHUFF64x2 zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	A	V/V	AVX512F OR AVX10.1	Shuffle 128-bit packed double precision floating-point values selected by imm8 from zmm2 and zmm3/m512/m64bcst and place results in zmm1 subject to writemask k1.
EVEX.256.66.0F3A.W0 43 /r ib VSHUFI32X4 ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shuffle 128-bit packed double-word values selected by imm8 from ymm2 and ymm3/m256/m32bcst and place results in ymm1 subject to writemask k1.
EVEX.512.66.0F3A.W0 43 /r ib VSHUFI32x4 zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	A	V/V	AVX512F OR AVX10.1	Shuffle 128-bit packed double-word values selected by imm8 from zmm2 and zmm3/m512/m32bcst and place results in zmm1 subject to writemask k1.
EVEX.256.66.0F3A.W1 43 /r ib VSHUFI64X2 ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shuffle 128-bit packed quad-word values selected by imm8 from ymm2 and ymm3/m256/m64bcst and place results in ymm1 subject to writemask k1.
EVEX.512.66.0F3A.W1 43 /r ib VSHUFI64x2 zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	A	V/V	AVX512F OR AVX10.1	Shuffle 128-bit packed quad-word values selected by imm8 from zmm2 and zmm3/m512/m64bcst and place results in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

256-bit Version: Moves one of the two 128-bit packed single precision floating-point values from the first source operand (second operand) into the low 128-bit of the destination operand (first operand); moves one of the two packed 128-bit floating-point values from the second source operand (third operand) into the high 128-bit of the destination operand. The selector operand (third operand) determines which values are moved to the destination operand.

512-bit Version: Moves two of the four 128-bit packed single precision floating-point values from the first source operand (second operand) into the low 256-bit of each double qword of the destination operand (first operand); moves two of the four packed 128-bit floating-point values from the second source operand (third operand) into the high 256-bit of the destination operand. The selector operand (third operand) determines which values are moved to the destination operand.

The first source operand is a vector register. The second source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a vector register.

The writemask updates the destination operand with the granularity of 32/64-bit data elements.

Operation

```
Select2(SRC, control) {
CASE (control[0]) OF
    0:    TMP := SRC[127:0];
    1:    TMP := SRC[255:128];
ESAC;
RETURN TMP
}

Select4(SRC, control) {
CASE (control[1:0]) OF
    0:    TMP := SRC[127:0];
    1:    TMP := SRC[255:128];
    2:    TMP := SRC[383:256];
    3:    TMP := SRC[511:384];
ESAC;
RETURN TMP
}
```

VSHUFF32x4 (EVEX versions)

```
(KL, VL) = (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] := SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i]
    FI;
ENDFOR;
IF VL = 256
    TMP_DEST[127:0] := Select2(SRC1[255:0], imm8[0]);
    TMP_DEST[255:128] := Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
    TMP_DEST[127:0] := Select4(SRC1[511:0], imm8[1:0]);
    TMP_DEST[255:128] := Select4(SRC1[511:0], imm8[3:2]);
    TMP_DEST[383:256] := Select4(TMP_SRC2[511:0], imm8[5:4]);
    TMP_DEST[511:384] := Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    THEN DEST[i+31:i] := 0
            FI;
    FI;
```

```
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VSHUFF64x2 (EVEX 512-bit version)

```
(KL, VL) = (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i]
    FI;
ENDFOR;
IF VL = 256
    TMP_DEST[127:0] := Select2(SRC1[255:0], imm8[0]);
    TMP_DEST[255:128] := Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
    TMP_DEST[127:0] := Select4(SRC1[511:0], imm8[1:0]);
    TMP_DEST[255:128] := Select4(SRC1[511:0], imm8[3:2]);
    TMP_DEST[383:256] := Select4(TMP_SRC2[511:0], imm8[5:4]);
    TMP_DEST[511:384] := Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    THEN DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VSHUFI32x4 (EVEX 512-bit version)

```
(KL, VL) = (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] := SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i]
    FI;
ENDFOR;
IF VL = 256
    TMP_DEST[127:0] := Select2(SRC1[255:0], imm8[0]);
    TMP_DEST[255:128] := Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
    TMP_DEST[127:0] := Select4(SRC1[511:0], imm8[1:0]);
    TMP_DEST[255:128] := Select4(SRC1[511:0], imm8[3:2]);
    TMP_DEST[383:256] := Select4(TMP_SRC2[511:0], imm8[5:4]);
    TMP_DEST[511:384] := Select4(TMP_SRC2[511:0], imm8[7:6]);
```

```
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    THEN DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VSHUFI64x2 (EVEX 512-bit version)

```
(KL, VL) = (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i]
    FI;
ENDFOR;
IF VL = 256
    TMP_DEST[127:0] := Select2(SRC1[255:0], imm8[0]);
    TMP_DEST[255:128] := Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
    TMP_DEST[127:0] := Select4(SRC1[511:0], imm8[1:0]);
    TMP_DEST[255:128] := Select4(SRC1[511:0], imm8[3:2]);
    TMP_DEST[383:256] := Select4(TMP_SRC2[511:0], imm8[5:4]);
    TMP_DEST[511:384] := Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    THEN DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VSHUFI32x4 __m512i _mm512_shuffle_i32x4(__m512i a, __m512i b, int imm);
VSHUFI32x4 __m512i _mm512_mask_shuffle_i32x4(__m512i s, __mmask16 k, __m512i a, __m512i b, int imm);
VSHUFI32x4 __m512i _mm512_maskz_shuffle_i32x4(__mmask16 k, __m512i a, __m512i b, int imm);
VSHUFI32x4 __m256i _mm256_shuffle_i32x4(__m256i a, __m256i b, int imm);
VSHUFI32x4 __m256i _mm256_mask_shuffle_i32x4(__m256i s, __mmask8 k, __m256i a, __m256i b, int imm);
VSHUFI32x4 __m256i _mm256_maskz_shuffle_i32x4(__mmask8 k, __m256i a, __m256i b, int imm);
VSHUFF32x4 __m512 _mm512_shuffle_f32x4(__m512 a, __m512 b, int imm);
VSHUFF32x4 __m512 _mm512_mask_shuffle_f32x4(__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VSHUFF32x4 __m512 _mm512_maskz_shuffle_f32x4(__mmask16 k, __m512 a, __m512 b, int imm);
VSHUFI64x2 __m512i _mm512_shuffle_i64x2(__m512i a, __m512i b, int imm);
VSHUFI64x2 __m512i _mm512_mask_shuffle_i64x2(__m512i s, __mmask8 k, __m512i b, __m512i b, int imm);
VSHUFI64x2 __m512i _mm512_maskz_shuffle_i64x2(__mmask8 k, __m512i a, __m512i b, int imm);
VSHUFF64x2 __m512d _mm512_shuffle_f64x2(__m512d a, __m512d b, int imm);
VSHUFF64x2 __m512d _mm512_mask_shuffle_f64x2(__m512d s, __mmask8 k, __m512d a, __m512d b, int imm);
VSHUFF64x2 __m512d _mm512_maskz_shuffle_f64x2(__mmask8 k, __m512d a, __m512d b, int imm);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-52, "Type E4NF Class Exception Conditions."
Additionally:

#UD	If EVEX.L'L = 0 for VSHUFF32x4/VSHUFF64x2.

VSM3MSG1—Perform Initial Calculation for the Next Four SM3 Message Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.NP.0F38.W0 DA /r VSM3MSG1 xmm1, xmm2, xmm3/m128	A	V/V	AVX SM3	Performs an initial calculation for the next four SM3 message words using previous message words from xmm2 and xmm3/m128, storing the result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

The VSM3MSG1 instruction is one of the two SM3 message scheduling instructions. The instruction performs an initial calculation for the next four SM3 message words.

Operation

```
define ROL32(dword, n):
    count := n % 32
    dest := (dword << count) | (dword >> (32-count))
    return dest

define P1(x):
    return x ^ ROL32(x, 15) ^ ROL32(x, 23)

VSM3MSG1 SRCDEST, SRC1, SRC2
W[0] := SRC2.dword[0]
W[1] := SRC2.dword[1]
W[2] := SRC2.dword[2]
W[3] := SRC2.dword[3]

W[7] := SRCDEST.dword[0]
W[8] := SRCDEST.dword[1]
W[9] := SRCDEST.dword[2]
W[10] := SRCDEST.dword[3]

W[13] := SRC1.dword[0]
W[14] := SRC1.dword[1]
W[15] := SRC1.dword[2]

TMP0 := W[7] ^ W[0] ^ ROL32(W[13], 15)
TMP1 := W[8] ^ W[1] ^ ROL32(W[14], 15)
TMP2 := W[9] ^ W[2] ^ ROL32(W[15], 15)
TMP3 := W[10] ^ W[3]

SRCDEST.dword[0] := P1(TMP0)
SRCDEST.dword[1] := P1(TMP1)
SRCDEST.dword[2] := P1(TMP2)
SRCDEST.dword[3] := P1(TMP3)
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VSM3MSG1 __m128i _mm_sm3msg1_epi32 (__m128i __A, __m128i __B, __m128i __C);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions."

VSM3MSG2—Perform Final Calculation for the Next Four SM3 Message Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 DA /r VSM3MSG2 xmm1, xmm2, xmm3/m128	A	V/V	AVX SM3	Performs the final calculation for the next four SM3 message words using previous message words from xmm2 and xmm3/m128, storing the result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

The VSM3MSG2 instruction is one of the two SM3 message scheduling instructions. The instruction performs the final calculation for the next four SM3 message words.

Operation

//see the VSM3MSG1 instruction for definition of ROL32()

VSM3MSG2 SRCDEST, SRC1, SRC2

```
WTMP[0] := SRCDEST.dword[0]
WTMP[1] := SRCDEST.dword[1]
WTMP[2] := SRCDEST.dword[2]
WTMP[3] := SRCDEST.dword[3]

// Dword array W[] has indices are based on the SM3 specification.
W[3] := SRC1.dword[0]
W[4] := SRC1.dword[1]
W[5] := SRC1.dword[2]
W[6] := SRC1.dword[3]
W[10] := SRC2.dword[0]
W[11] := SRC2.dword[1]
W[12] := SRC2.dword[2]
W[13] := SRC2.dword[3]

W[16] := ROL32(W[3], 7) ^ W[10] ^ WTMP[0]
W[17] := ROL32(W[4], 7) ^ W[11] ^ WTMP[1]
W[18] := ROL32(W[5], 7) ^ W[12] ^ WTMP[2]
W[19] := ROL32(W[6], 7) ^ W[13] ^ WTMP[3]

W[19] := W[19] ^ ROL32(W[16], 6) ^ ROL32(W[16], 15) ^ ROL32(W[16], 30)

SRCDEST.dword[0] := W[16]
SRCDEST.dword[1] := W[17]
SRCDEST.dword[2] := W[18]
SRCDEST.dword[3] := W[19]
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VSM3MSG2 __m128i _mm_sm3msg2_epi32 (__m128i __A, __m128i __B, __m128i __C);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions."

VSM3RNDS2—Perform Two Rounds of SM3 Operation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F3A.W0 DE /r /ib VSM3RNDS2 xmm1, xmm2, xmm3/m128, imm8	A	V/V	AVX SM3	Performs two rounds of SM3 operation using the initial SM3 states from xmm1 and xmm2, and pre-computed words from xmm3/m128, storing the result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

The VSM3RNDS2 instruction performs two rounds of SM3 operation using initial SM3 state (C, D, G, H) from the first operand, an initial SM3 states (A, B, E, F) from the second operand and a pre-computed words from the third operand. The first operand with initial SM3 state of (C, D, G, H) assumes input of non-rotated left variables from previous state. The updated SM3 state (A, B, E, F) is written to the first operand.

The imm8 should contain the even round number for the first of the two rounds computed by this instruction. The computation masks the imm8 value by AND'ing it with 0x3E so that only even round numbers from 0 through 62 are used for this operation.

Operation

```
//see the VSM3MSG1 instruction for definition of ROL32()

define P0(dword):
    return dword ^ ROL32(dword, 9) ^ ROL32(dword, 17)

define FF(x,y,z, round):
    if round < 16:
        return (x ^ y ^ z)
    else:
        return (x & y) | (x & z) | (y & z)

define GG(x,y,z, round):
    if round < 16:
        return (x ^ y ^ z)
    else:
        return (x & y) | (~x & z)
```

VSM3RNDS2 SRCDEST, SRC1, SRC2, IMM8

```
A[0] := SRC1.dword[3]
B[0] := SRC1.dword[2]
C[0] := SRCDEST.dword[3]
D[0] := SRCDEST.dword[2]
E[0] := SRC1.dword[1]
F[0] := SRC1.dword[0]
G[0] := SRCDEST.dword[1]
H[0] := SRCDEST.dword[0]
W[0] := SRC2.dword[0]
W[1] := SRC2.dword[1]
W[4] := SRC2.dword[2]
```

```
W[5] := SRC2.dword[3]

C[0] := ROL32(C[0], 9)
D[0] := ROL32(D[0], 9)
G[0] := ROL32(G[0], 19)
H[0] := ROL32(H[0], 19)

ROUND := IMM8 & 0x3E // even numbers 0...62
IF ROUND < 16:
    CONST := 0x79cc4519
ELSE:
    CONST := 0x7a879d8a
CONST := ROL32(CONST,ROUND)

FOR i in 0..1:
    S1 := ROL32((ROL32(A[i], 12) + E[i] + CONST), 7)
    S2 := S1 ^ ROL32(A[i],12)
    T1 := FF(A[i], B[i], C[i], ROUND) + D[i] + S2 + (W[i]^W[i+4])
    T2 := GG(E[i], F[i], G[i], ROUND) + H[i] + S1 + W[i]
    D[i+1] := C[i]
    C[i+1] := ROL32(B[i],9)
    B[i+1] := A[i]
    A[i+1] := T1
    H[i+1] := G[i]
    G[i+1] := ROL32(F[i], 19)
    F[i+1] := E[i]
    E[i+1] := P0(T2)
    CONST := ROL32(CONST, 1)
SRCDEST.dword[3] := A[2]
SRCDEST.dword[2] := B[2]
SRCDEST.dword[1] := E[2]
SRCDEST.dword[0] := F[2]
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VSM3RNDS2 __m128i _mm_sm3rnds2_epi32 (__m128i __A, __m128i __B, __m128i __C, const int imm8);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions."

VSM4KEY4—Perform Four Rounds of SM4 Key Expansion

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.F3.0F38.W0 DA /r VSM4KEY4 xmm1, xmm2, xmm3/m128	A	V/V	AVX SM4	Performs four rounds of SM4 key expansion.
VEX.256.F3.0F38.W0 DA /r VSM4KEY4 ymm1, ymm2, ymm3/m256	A	V/V	AVX SM4	Performs four rounds of SM4 key expansion.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

The VSM4KEY4 instruction performs four rounds of SM4 key expansion. The instruction operates on independent 128-bit lanes.

Additional details can be found at: https://tools.ietf.org/html/draft-ribose-cfrg-sm4-10.

Both SM4 instructions use a common sbox table:

```
BYTE sbox[256] = {
0xD6, 0x90, 0xE9, 0xFE, 0xCC, 0xE1, 0x3D, 0xB7, 0x16, 0xB6, 0x14, 0xC2, 0x28, 0xFB, 0x2C, 0x05,
0x2B, 0x67, 0x9A, 0x76, 0x2A, 0xBE, 0x04, 0xC3, 0xAA, 0x44, 0x13, 0x26, 0x49, 0x86, 0x06, 0x99,
0x9C, 0x42, 0x50, 0xF4, 0x91, 0xEF, 0x98, 0x7A, 0x33, 0x54, 0x0B, 0x43, 0xED, 0xCF, 0xAC, 0x62,
0xE4, 0xB3, 0x1C, 0xA9, 0xC9, 0x08, 0xE8, 0x95, 0x80, 0xDF, 0x94, 0xFA, 0x75, 0x8F, 0x3F, 0xA6,
0x47, 0x07, 0xA7, 0xFC, 0xF3, 0x73, 0x17, 0xBA, 0x83, 0x59, 0x3C, 0x19, 0xE6, 0x85, 0x4F, 0xA8,
0x68, 0x6B, 0x81, 0xB2, 0x71, 0x64, 0xDA, 0x8B, 0xF8, 0xEB, 0x0F, 0x4B, 0x70, 0x56, 0x9D, 0x35,
0x1E, 0x24, 0x0E, 0x5E, 0x63, 0x58, 0xD1, 0xA2, 0x25, 0x22, 0x7C, 0x3B, 0x01, 0x21, 0x78, 0x87,
0xD4, 0x00, 0x46, 0x57, 0x9F, 0xD3, 0x27, 0x52, 0x4C, 0x36, 0x02, 0xE7, 0xA0, 0xC4, 0xC8, 0x9E,
0xEA, 0xBF, 0x8A, 0xD2, 0x40, 0xC7, 0x38, 0xB5, 0xA3, 0xF7, 0xF2, 0xCE, 0xF9, 0x61, 0x15, 0xA1,
0xE0, 0xAE, 0x5D, 0xA4, 0x9B, 0x34, 0x1A, 0x55, 0xAD, 0x93, 0x32, 0x30, 0xF5, 0x8C, 0xB1, 0xE3,
0x1D, 0xF6, 0xE2, 0x2E, 0x82, 0x66, 0xCA, 0x60, 0xC0, 0x29, 0x23, 0xAB, 0x0D, 0x53, 0x4E, 0x6F,
0xD5, 0xDB, 0x37, 0x45, 0xDE, 0xFD, 0x8E, 0x2F, 0x03, 0xFF, 0x6A, 0x72, 0x6D, 0x6C, 0x5B, 0x51,
0x8D, 0x1B, 0xAF, 0x92, 0xBB, 0xDD, 0xBC, 0x7F, 0x11, 0xD9, 0x5C, 0x41, 0x1F, 0x10, 0x5A, 0xD8,
0x0A, 0xC1, 0x31, 0x88, 0xA5, 0xCD, 0x7B, 0xBD, 0x2D, 0x74, 0xD0, 0x12, 0xB8, 0xE5, 0xB4, 0xB0,
0x89, 0x69, 0x97, 0x4A, 0x0C, 0x96, 0x77, 0x7E, 0x65, 0xB9, 0xF1, 0x09, 0xC5, 0x6E, 0xC6, 0x84,
0x18, 0xF0, 0x7D, 0xEC, 0x3A, 0xDC, 0x4D, 0x20, 0x79, 0xEE, 0x5F, 0x3E, 0xD7, 0xCB, 0x39, 0x48
}
```

Operation

```
define ROL32(dword, n):
    count := n % 32
    dest := (dword << count) | (dword >> (32-count))
    return dest

define SBOX_BYTE(dword, i):
    // sbox[] array defined in introduction
    return sbox[dword.byte[i]]

define lower_t(dword):
    tmp.byte[0] := SBOX_BYTE(dword, 0)
```

```
    tmp.byte[1] := SBOX_BYTE(dword, 1)
    tmp.byte[2] := SBOX_BYTE(dword, 2)
    tmp.byte[3] := SBOX_BYTE(dword, 3)
    return tmp

define L_KEY(dword):
    return dword ^ ROL32(dword, 13) ^ ROL32(dword, 23)

define T_KEY(dword):
    return L_KEY(lower_t(dword))

define F_KEY(X0, X1, X2, X3, round_key):
    return X0 ^ T_KEY(X1 ^ X2 ^ X3 ^ round_key)
```

VSM4KEY4 DEST, SRC1, SRC2

```
VL = (128, 256) // VEX versions
// or
VL = (128,256,512) // EVEX versions
KL := VL/128

for i in 0..KL-1:
    P[0] := SRC1.xmm[i].dword[0]
    P[1] := SRC1.xmm[i].dword[1]
    P[2] := SRC1.xmm[i].dword[2]
    P[3] := SRC1.xmm[i].dword[3]

    C[0] := F_KEY(P[0], P[1], P[2], P[3], SRC2.xmm[i].dword[0])
    C[1] := F_KEY(P[1], P[2], P[3], C[0], SRC2.xmm[i].dword[1])
    C[2] := F_KEY(P[2], P[3], C[0], C[1], SRC2.xmm[i].dword[2])
    C[3] := F_KEY(P[3], C[0], C[1], C[2], SRC2.xmm[i].dword[3])

    DEST.xmm[i].dword[0] := C[0]
    DEST.xmm[i].dword[1] := C[1]
    DEST.xmm[i].dword[2] := C[2]
    DEST.xmm[i].dword[3] := C[3]

DEST[MAXVL-1:VL] := 0
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VSM4KEY4 __m128i _mm_sm4key4_epi32 (__m128i __A, __m128i __B);
VSM4KEY4 __m256i _mm256_sm4key4_epi32 (__m256i __A, __m256i __B);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-23, “Type 6 Class Exception Conditions.”

VSM4RNDS4—Performs Four Rounds of SM4 Encryption

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.F2.0F38.W0 DA /r VSM4RNDS4 xmm1, xmm2, xmm3/m128	A	V/V	AVX SM4	Performs four rounds of SM4 encryption.
VEX.256.F2.0F38.W0 DA /r VSM4RNDS4 ymm1, ymm2, ymm3/m256	A	V/V	AVX SM4	Performs four rounds of SM4 encryption.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

The SM4RNDS4 instruction performs four rounds of SM4 encryption. The instruction operates on independent 128-bit lanes.

Additional details can be found at: https://tools.ietf.org/html/draft-ribose-cfrg-sm4-10.

See "VSM4KEY4—Perform Four Rounds of SM4 Key Expansion" for the sbox table.

Operation

```
// see the VSM4KEY4 instruction for the definition of ROL32, lower_t

define L_RND(dword):
    tmp := dword
    tmp := tmp ^ ROL32(dword, 2)
    tmp := tmp ^ ROL32(dword, 10)
    tmp := tmp ^ ROL32(dword, 18)
    tmp := tmp ^ ROL32(dword, 24)
    return tmp

define T_RND(dword):
    return L_RND(lower_t(dword))

define F_RND(X0, X1, X2, X3, round_key):
    return X0 ^ T_RND(X1 ^ X2 ^ X3 ^ round_key)
```

VSM4RNDS4 DEST, SRC1, SRC2

```
VL = (128, 256) // VEX versions
KL := VL/128

for i in 0..KL-1:
    P[0] := SRC1.xmm[i].dword[0]
    P[1] := SRC1.xmm[i].dword[1]
    P[2] := SRC1.xmm[i].dword[2]
    P[3] := SRC1.xmm[i].dword[3]

    C[0] := F_RND(P[0], P[1], P[2], P[3], SRC2.xmm[i].dword[0])
    C[1] := F_RND(P[1], P[2], P[3], C[0], SRC2.xmm[i].dword[1])
    C[2] := F_RND(P[2], P[3], C[0], C[1], SRC2.xmm[i].dword[2])
    C[3] := F_RND(P[3], C[0], C[1], C[2], SRC2.xmm[i].dword[3])

    DEST.xmm[i].dword[0] := C[0]
    DEST.xmm[i].dword[1] := C[1]
    DEST.xmm[i].dword[2] := C[2]
    DEST.xmm[i].dword[3] := C[3]

DEST[MAXVL-1:VL] := 0
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VSM4RNDS4 __m128i _mm_sm4rnds4_epi32 (__m128i __A, __m128i __B);
VSM4RNDS4 __m256i _mm256_sm4rnds4_epi32 (__m256i __A, __m256i __B);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-23, "Type 6 Class Exception Conditions."

VSQRTPH—Compute Square Root of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 51 /r VSQRTPH xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Compute square roots of the packed FP16 values in xmm2/m128/m16bcst, and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 51 /r VSQRTPH ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Compute square roots of the packed FP16 values in ymm2/m256/m16bcst, and store the result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 51 /r VSQRTPH zmm1{k1}{z}, zmm2/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1	Compute square roots of the packed FP16 values in zmm2/m512/m16bcst, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction performs a packed FP16 square-root computation on the values from source operand and stores the packed FP16 result in the destination operand. The destination elements are updated according to the write-mask.

Operation

VSQRTPH dest{k1}, src

```
VL = 128, 256 or 512
KL := VL/16

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.fp16[0]
        ELSE:
            tsrc := src.fp16[i]
        DEST.fp16[i] := SQRT(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[i] := 0
    //else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VSQRTPH __m128h _mm_mask_sqrt_ph (__m128h src, __mmask8 k, __m128h a);
VSQRTPH __m128h _mm_maskz_sqrt_ph (__mmask8 k, __m128h a);
VSQRTPH __m128h _mm_sqrt_ph (__m128h a);
VSQRTPH __m256h _mm256_mask_sqrt_ph (__m256h src, __mmask16 k, __m256h a);
VSQRTPH __m256h _mm256_maskz_sqrt_ph (__mmask16 k, __m256h a);
VSQRTPH __m256h _mm256_sqrt_ph (__m256h a);
VSQRTPH __m512h _mm512_mask_sqrt_ph (__m512h src, __mmask32 k, __m512h a);
VSQRTPH __m512h _mm512_maskz_sqrt_ph (__mmask32 k, __m512h a);
VSQRTPH __m512h _mm512_sqrt_ph (__m512h a);
VSQRTPH __m512h _mm512_mask_sqrt_round_ph (__m512h src, __mmask32 k, __m512h a, const int rounding);
VSQRTPH __m512h _mm512_maskz_sqrt_round_ph (__mmask32 k, __m512h a, const int rounding);
VSQRTPH __m512h _mm512_sqrt_round_ph (__m512h a, const int rounding);

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Other Exceptions

EVEX-encoded instruction, see Table 2-48, “Type E2 Class Exception Conditions.”

VSQRTSH—Compute Square Root of Scalar FP16 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 51 /r VSQRTSH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512_FP16 OR AVX10.1	Compute square root of the low FP16 value in xmm3/m16 and store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a scalar FP16 square-root computation on the source operand and stores the FP16 result in the destination operand. Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

VSQRTSH dest{k1}, src1, src2

```
IF k1[0] or *no writemask*:
    DEST.fp16[0] := SQRT(src2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
//else DEST.fp16[0] remains unchanged

DEST[127:16] := src1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VSQRTSH __m128h _mm_mask_sqrt_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, const int rounding);
VSQRTSH __m128h _mm_maskz_sqrt_round_sh (__mmask8 k, __m128h a, __m128h b, const int rounding);
VSQRTSH __m128h _mm_sqrt_round_sh (__m128h a, __m128h b, const int rounding);
VSQRTSH __m128h _mm_mask_sqrt_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VSQRTSH __m128h _mm_maskz_sqrt_sh (__mmask8 k, __m128h a, __m128h b);
VSQRTSH __m128h _mm_sqrt_sh (__m128h a, __m128h b);

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VSUBPH—Subtract Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 5C /r VSUBPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1	Subtract packed FP16 values from xmm3/m128/m16bcst to xmm2, and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 5C /r VSUBPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512_FP16 AND AVX512VL) OR AVX10.1[1]	Subtract packed FP16 values from ymm3/m256/m16bcst to ymm2, and store the result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 5C /r VSUBPH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512_FP16 OR AVX10.1[1]	Subtract packed FP16 values from zmm3/m512/m16bcst to zmm2, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction subtracts packed FP16 values from second source operand from the corresponding elements in the first source operand, storing the packed FP16 result in the destination operand. The destination elements are updated according to the writemask.

Operation

VSUBPH (EVEX encoded versions) when src2 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.fp16[j] := SRC1.fp16[j] - SRC2.fp16[j]
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VSUBPH (EVEX encoded versions) when src2 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            DEST.fp16[j] := SRC1.fp16[j] - SRC2.fp16[0]
        ELSE:
            DEST.fp16[j] := SRC1.fp16[j] - SRC2.fp16[j]
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSUBPH __m128h _mm_mask_sub_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VSUBPH __m128h _mm_maskz_sub_ph (__mmask8 k, __m128h a, __m128h b);
VSUBPH __m128h _mm_sub_ph (__m128h a, __m128h b);
VSUBPH __m256h _mm256_mask_sub_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VSUBPH __m256h _mm256_maskz_sub_ph (__mmask16 k, __m256h a, __m256h b);
VSUBPH __m256h _mm256_sub_ph (__m256h a, __m256h b);
VSUBPH __m512h _mm512_mask_sub_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VSUBPH __m512h _mm512_maskz_sub_ph (__mmask32 k, __m512h a, __m512h b);
VSUBPH __m512h _mm512_sub_ph (__m512h a, __m512h b);
VSUBPH __m512h _mm512_mask_sub_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int rounding);
VSUBPH __m512h _mm512_maskz_sub_round_ph (__mmask32 k, __m512h a, __m512h b, int rounding);
VSUBPH __m512h _mm512_sub_round_ph (__m512h a, __m512h b, int rounding);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instruction, see Table 2-48, “Type E2 Class Exception Conditions.”

VSUBSH—Subtract Scalar FP16 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 5C /r VSUBSH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512_FP16 OR AVX10.1	Subtract the low FP16 value in xmm3/m16 from xmm2 and store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction subtracts the low FP16 value from the second source operand from the corresponding value in the first source operand, storing the FP16 result in the destination operand. Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

VSUBSH (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC2 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    DEST.fp16[0] := SRC1.fp16[0] - SRC2.fp16[0]
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else dest.fp16[0] remains unchanged
DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VSUBSH __m128h _mm_mask_sub_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int rounding);
VSUBSH __m128h _mm_maskz_sub_round_sh (__mmask8 k, __m128h a, __m128h b, int rounding);
VSUBSH __m128h _mm_sub_round_sh (__m128h a, __m128h b, int rounding);
VSUBSH __m128h _mm_mask_sub_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VSUBSH __m128h _mm_maskz_sub_sh (__mmask8 k, __m128h a, __m128h b);
VSUBSH __m128h _mm_sub_sh (__m128h a, __m128h b);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VTESTPD/VTESTPS—Packed Bit Test

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 0E /r VTESTPS xmm1, xmm2/m128	RM	V/V	AVX	Set ZF and CF depending on sign bit AND and ANDN of packed single precision floating-point sources.
VEX.256.66.0F38.W0 0E /r VTESTPS ymm1, ymm2/m256	RM	V/V	AVX	Set ZF and CF depending on sign bit AND and ANDN of packed single precision floating-point sources.
VEX.128.66.0F38.W0 0F /r VTESTPD xmm1, xmm2/m128	RM	V/V	AVX	Set ZF and CF depending on sign bit AND and ANDN of packed double precision floating-point sources.
VEX.256.66.0F38.W0 0F /r VTESTPD ymm1, ymm2/m256	RM	V/V	AVX	Set ZF and CF depending on sign bit AND and ANDN of packed double precision floating-point sources.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	N/A	N/A

Description

VTESTPS performs a bitwise comparison of all the sign bits of the packed single precision elements in the first source operation and corresponding sign bits in the second source operand. If the AND of the source sign bits with the dest sign bits produces all zeros, the ZF is set else the ZF is clear. If the AND of the source sign bits with the inverted dest sign bits produces all zeros the CF is set else the CF is clear. An attempt to execute VTESTPS with VEX.W=1 will cause #UD.

VTESTPD performs a bitwise comparison of all the sign bits of the double precision elements in the first source operation and corresponding sign bits in the second source operand. If the AND of the source sign bits with the dest sign bits produces all zeros, the ZF is set else the ZF is clear. If the AND the source sign bits with the inverted dest sign bits produces all zeros the CF is set else the CF is clear. An attempt to execute VTESTPS with VEX.W=1 will cause #UD.

The first source register is specified by the ModR/M *reg* field.

128-bit version: The first source register is an XMM register. The second source register can be an XMM register or a 128-bit memory location. The destination register is not modified.

VEX.256 encoded version: The first source register is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination register is not modified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VTESTPS (128-bit version)

```
TEMP[127:0] := SRC[127:0] AND DEST[127:0]
IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = 0)
    THEN ZF := 1;
    ELSE ZF := 0;

TEMP[127:0] := SRC[127:0] AND NOT DEST[127:0]
IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = 0)
    THEN CF := 1;
    ELSE CF := 0;
DEST (unmodified)
AF := OF := PF := SF := 0;
```

VTESTPS (VEX.256 encoded version)

```
TEMP[255:0] := SRC[255:0] AND DEST[255:0]
IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127]= TEMP[160] =TEMP[191] = TEMP[224] = TEMP[255] = 0)
    THEN ZF := 1;
    ELSE ZF := 0;

TEMP[255:0] := SRC[255:0] AND NOT DEST[255:0]
IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127]= TEMP[160] =TEMP[191] = TEMP[224] = TEMP[255] = 0)
    THEN CF := 1;
    ELSE CF := 0;
DEST (unmodified)
AF := OF := PF := SF := 0;
```

VTESTPD (128-bit version)

```
TEMP[127:0] := SRC[127:0] AND DEST[127:0]
IF ( TEMP[63] = TEMP[127] = 0)
    THEN ZF := 1;
    ELSE ZF := 0;

TEMP[127:0] := SRC[127:0] AND NOT DEST[127:0]
IF ( TEMP[63] = TEMP[127] = 0)
    THEN CF := 1;
    ELSE CF := 0;
DEST (unmodified)
AF := OF := PF := SF := 0;
```

VTESTPD (VEX.256 encoded version)

```
TEMP[255:0] := SRC[255:0] AND DEST[255:0]
IF (TEMP[63] = TEMP[127] = TEMP[191] = TEMP[255] = 0)
    THEN ZF := 1;
    ELSE ZF := 0;

TEMP[255:0] := SRC[255:0] AND NOT DEST[255:0]
IF (TEMP[63] = TEMP[127] = TEMP[191] = TEMP[255] = 0)
    THEN CF := 1;
    ELSE CF := 0;
DEST (unmodified)
AF := OF := PF := SF := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VTESTPS

int _mm256_testz_ps (__m256 s1, __m256 s2);

int _mm256_testc_ps (__m256 s1, __m256 s2);

int _mm256_testnzc_ps (__m256 s1, __m128 s2);

int _mm_testz_ps (__m128 s1, __m128 s2);

int _mm_testc_ps (__m128 s1, __m128 s2);

int _mm_testnzc_ps (__m128 s1, __m128 s2);

VTESTPD

int _mm256_testz_pd (__m256d s1, __m256d s2);

int _mm256_testc_pd (__m256d s1, __m256d s2);

int _mm256_testnzc_pd (__m256d s1, __m256d s2);

int _mm_testz_pd (__m128d s1, __m128d s2);

int _mm_testc_pd (__m128d s1, __m128d s2);

int _mm_testnzc_pd (__m128d s1, __m128d s2);

Flags Affected

The OF, AF, PF, SF flags are cleared and the ZF, CF flags are set according to the operation.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions."

Additionally:

#UD	If VEX.vvvv ≠ 1111B.
	If VEX.W = 1 for VTESTPS or VTESTPD.

VUCOMISH—Unordered Compare Scalar FP16 Values and Set EFLAGS

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.NP.MAP5.W0 2E /r VUCOMISH xmm1, xmm2/m16 {sae}	A	V/V	AVX512_FP16 OR AVX10.1	Compare low FP16 values in xmm1 and xmm2/m16 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction compares the FP16 values in the low word of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 16-bit memory location.

The VUCOMISH instruction differs from the VCOMISH instruction in that it signals a SIMD floating-point invalid operation exception (#I) only if a source operand is an SNaN. The COMISS instruction signals an invalid numeric exception when a source operand is either a QNaN or SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated. EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Operation

VUCOMISH

```
RESULT := UnorderedCompare(SRC1.fp16[0],SRC2.fp16[0])
if RESULT is UNORDERED:
    ZF, PF, CF := 1, 1, 1
else if RESULT is GREATER_THAN:
    ZF, PF, CF := 0, 0, 0
else if RESULT is LESS_THAN:
    ZF, PF, CF := 0, 0, 1
else: // RESULT is EQUALS
    ZF, PF, CF := 1, 0, 0

OF, AF, SF := 0, 0, 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VUCOMISH int _mm_ucomieq_sh (__m128h a, __m128h b);
VUCOMISH int _mm_ucomige_sh (__m128h a, __m128h b);
VUCOMISH int _mm_ucomigt_sh (__m128h a, __m128h b);
VUCOMISH int _mm_ucomile_sh (__m128h a, __m128h b);
VUCOMISH int _mm_ucomilt_sh (__m128h a, __m128h b);
VUCOMISH int _mm_ucomineq_sh (__m128h a, __m128h b);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-50, "Type E3NF Class Exception Conditions."

VZEROALL—Zero XMM, YMM, and ZMM Registers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.0F.WIG 77 VZEROALL	ZO	V/V	AVX	Zero some of the XMM, YMM, and ZMM registers.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

In 64-bit mode, the instruction zeroes XMM0-XMM15, YMM0-YMM15, and ZMM0-ZMM15. Outside 64-bit mode, it zeroes only XMM0-XMM7, YMM0-YMM7, and ZMM0-ZMM7. VZEROALL does not modify ZMM16-ZMM31.

Note: VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD. In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

Operation

simd_reg_file[][] is a two dimensional array representing the SIMD register file containing all the overlapping xmm, ymm, and zmm registers present in that implementation. The major dimension is the register number: 0 for xmm0, ymm0, and zmm0; 1 for xmm1, ymm1, and zmm1; etc. The minor dimension size is the width of the implemented SIMD state measured in bits. On a machine supporting Intel AVX-512, the width is 512.

VZEROALL (VEX.256 encoded version)

```
IF (64-bit mode)
    limit :=15
ELSE
    limit := 7
FOR i in 0 .. limit:
    simd_reg_file[i][MAXVL-1:0] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VZEROALL: _mm256_zeroall()

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-25, "Type 8 Class Exception Conditions."

VZEROUPPER—Zero Upper Bits of YMM and ZMM Registers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.0F.WIG 77 VZEROUPPER	ZO	V/V	AVX	Zero bits in positions 128 and higher of some YMM and ZMM registers.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

In 64-bit mode, the instruction zeroes the bits in positions 128 and higher in YMM0-YMM15 and ZMM0-ZMM15. Outside 64-bit mode, it zeroes those bits only in YMM0-YMM7 and ZMM0-ZMM7. VZEROUPPER does not modify the lower 128 bits of these registers and it does not modify ZMM16-ZMM31.

This instruction is recommended when transitioning between AVX and legacy SSE code; it will eliminate performance penalties caused by false dependencies.

Note: VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD. In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

Operation

simd_reg_file[][] is a two dimensional array representing the SIMD register file containing all the overlapping xmm, ymm, and zmm registers present in that implementation. The major dimension is the register number: 0 for xmm0, ymm0, and zmm0; 1 for xmm1, ymm1, and zmm1; etc. The minor dimension size is the width of the implemented SIMD state measured in bits.

VZEROUPPER

```
IF (64-bit mode)
    limit :=15
ELSE
    limit := 7
FOR i in 0 .. limit:
    simd_reg_file[i][MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VZEROUPPER: _mm256_zeroupper()

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-25, "Type 8 Class Exception Conditions."

CHAPTER 6
INSTRUCTION SET REFERENCE, W-Z

6.1 INSTRUCTIONS (W-Z)

Chapter 6 continues an alphabetical discussion of Intel® 64 and IA-32 instructions (W-Z). See also: Chapter 3, "Instruction Set Reference, A-L," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A; Chapter 4, "Instruction Set Reference, M-U," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B; and Chapter 5, "Instruction Set Reference, V," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D.

WAIT/FWAIT—Wait

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
9B	WAIT	ZO	Valid	Valid	Check pending unmasked floating-point exceptions.
9B	FWAIT	ZO	Valid	Valid	Check pending unmasked floating-point exceptions.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Causes the processor to check for and handle pending, unmasked, floating-point exceptions before proceeding. (FWAIT is an alternate mnemonic for WAIT.)

This instruction is useful for synchronizing exceptions in critical sections of code. Coding a WAIT instruction after a floating-point instruction ensures that any unmasked floating-point exceptions the instruction may raise are handled before the processor can modify the instruction's results. See the section titled "Floating-Point Exception Synchronization" in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for more information on using the WAIT/FWAIT instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
CheckForPendingUnmaskedFloatingPointExceptions;
```

FPU Flags Affected

The C0, C1, C2, and C3 flags are undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM	If CR0.MP[bit 1] = 1 and CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

WBINVD—Write Back and Invalidate Cache

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 09	WBINVD	ZO	Valid	Valid	Write back and flush Internal caches; initiate writing-back and flushing of external caches.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Writes back all modified cache lines in the processor's internal cache to main memory and invalidates (flushes) the internal caches. The instruction then issues a special-function bus cycle that directs external caches to also write back modified data and another bus cycle to indicate that the external caches should be invalidated.

After executing this instruction, the processor does not wait for the external caches to complete their write-back and flushing operations before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache write-back and flush signals. The amount of time or cycles for WBINVD to complete will vary due to size and other factors of different cache hierarchies. As a consequence, the use of the WBINVD instruction can have an impact on logical processor interrupt/event response time. Additional information of WBINVD behavior in a cache hierarchy with hierarchical sharing topology can be found in Chapter 2 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

The WBINVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction. This instruction is also a serializing instruction (see "Serializing Instructions" in Chapter 11, "Multiple-Processor Management," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A).

In situations where cache coherency with main memory is not a concern, software can use the INVD instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility

The WBINVD instruction is implementation dependent, and its function may be implemented differently on future Intel 64 and IA-32 processors. The instruction is not supported on IA-32 processors earlier than the Intel486 processor.

Operation

```
WriteBack(InternalCaches);
Flush(InternalCaches);
SignalWriteBack(ExternalCaches);
SignalFlush(ExternalCaches);
Continue; (* Continue execution *)
void _wbinvd(void)
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	WBINVD cannot be executed at the virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

WBNOINVD—Write Back and Do Not Invalidate Cache

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 09 WBNOINVD	ZO	V/V	WBNOINVD	Write back and do not flush internal caches; initiate writing-back without flushing of external caches.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A	N/A

Description

The WBNOINVD instruction writes back all modified cache lines in the processor's internal cache to main memory but does not invalidate (flush) the internal caches.

After executing this instruction, the processor does not wait for the external caches to complete their write-back operation before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache write-back signal. The amount of time or cycles for WBNOINVD to complete will vary due to size and other factors of different cache hierarchies. As a consequence, the use of the WBNOINVD instruction can have an impact on logical processor interrupt/event response time.

The WBNOINVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction. This instruction is also a serializing instruction (see "Serializing Instructions" in Chapter 11, "Multiple-Processor Management," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A).

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
WriteBack(InternalCaches);
Continue; (* Continue execution *)
```

Intel C/C++ Compiler Intrinsic Equivalent

WBNOINVD void _wbnoinvd(void);

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	WBNOINVD cannot be executed at the virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

WRFSBASE/WRGSBASE—Write FS/GS Segment Base

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F AE /2 WRFSBASE r32	M	V/I	FSGSBASE	Load the FS base address with the 32-bit value in the source register.
F3 REX.W 0F AE /2 WRFSBASE r64	M	V/I	FSGSBASE	Load the FS base address with the 64-bit value in the source register.
F3 0F AE /3 WRGSBASE r32	M	V/I	FSGSBASE	Load the GS base address with the 32-bit value in the source register.
F3 REX.W 0F AE /3 WRGSBASE r64	M	V/I	FSGSBASE	Load the GS base address with the 64-bit value in the source register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

Description

Loads the FS or GS segment base address with the general-purpose register indicated by the modR/M:r/m field.

The source operand may be either a 32-bit or a 64-bit general-purpose register. The REX.W prefix indicates the operand size is 64 bits. If no REX.W prefix is used, the operand size is 32 bits; the upper 32 bits of the source register are ignored and upper 32 bits of the base address (for FS or GS) are cleared.

This instruction is supported only in 64-bit mode.

Operation

```
FS/GS segment base address := SRC;
```

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

```
WRFSBASE  void _writefsbase_u32( unsigned int );
WRFSBASE  _writefsbase_u64( unsigned __int64 );
WRGSBASE  void _writegsbase_u32( unsigned int );
WRGSBASE  _writegsbase_u64( unsigned __int64 );
```

Protected Mode Exceptions

#UD	The WRFSBASE and WRGSBASE instructions are not recognized in protected mode.

Real-Address Mode Exceptions

#UD	The WRFSBASE and WRGSBASE instructions are not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The WRFSBASE and WRGSBASE instructions are not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	The WRFSBASE and WRGSBASE instructions are not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.FSGSBASE[bit 16] = 0.
	If CPUID.07H.00H:EBX.FSGSBASE[0] = 0
#GP(0)	If the source register contains a non-canonical address.

WRMSR—Write to Model Specific Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 30	WRMSR	ZO	Valid	Valid	Write the value in EDX:EAX to MSR specified by ECX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Writes the contents of registers EDX:EAX into the 64-bit model specific register (MSR) specified in the ECX register. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The contents of the EDX register are copied to high-order 32 bits of the selected MSR and the contents of the EAX register are copied to low-order 32 bits of the MSR. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are ignored.) Undefined or reserved bits in an MSR should be set to values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write to bits in a reserved MSR.

When the WRMSR instruction is used to write to an MTRR, the TLBs are invalidated. This includes global entries (see Section 5.10.2, "Translation Lookaside Buffers (TLBs)" of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A).

MSRs control functions for testability, execution tracing, performance-monitoring and machine check errors. Chapter 2, "Model-Specific Registers (MSRs)," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4, lists all MSRs that can be written with this instruction and their addresses. Note that each processor family has its own set of MSRs.

The WRMSR instruction is a serializing instruction (see "Serializing Instructions" in Chapter 11, "Multiple-Processor Management," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A). Note that WRMSR to the IA32_TSC_DEADLINE MSR (MSR index 6E0H) and the X2APIC MSRs (MSR indices 802H to 83FH) are not serializing.

The CPUID instruction should be used to determine whether MSRs are supported (CPUID.01H:EDX[5] = 1) before using this instruction.

IA-32 Architecture Compatibility

The MSRs and the ability to read them with the WRMSR instruction were introduced into the IA-32 architecture with the Pentium processor. Execution of this instruction by an IA-32 processor earlier than the Pentium processor results in an invalid opcode exception #UD.

Operation

```
MSR[ECX] := EDX:EAX;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If the value in ECX specifies a reserved or unimplemented MSR address.
	If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX.
	If the source register contains a non-canonical address and ECX specifies one of the following MSRs: IA32_DS_AREA, IA32_FS_BASE, IA32_GS_BASE, IA32_KERNEL_GS_BASE, IA32_L-STAR, IA32_SYSENTER_EIP, IA32_SYSENTER_ESP.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If the value in ECX specifies a reserved or unimplemented MSR address.
	If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX.
	If the source register contains a non-canonical address and ECX specifies one of the following MSRs: IA32_DS_AREA, IA32_FS_BASE, IA32_GS_BASE, IA32_KERNEL_GS_BASE, IA32_L-STAR, IA32_SYSENTER_EIP, IA32_SYSENTER_ESP.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	The WRMSR instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

WRMSRLIST—Write List of Model Specific Registers

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 01 C6 WRMSRLIST	ZO	V/N.E.	MSRLIST	Write requested list of MSRs with the values specified in memory.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

This instruction writes a software-provided list of up to 64 MSRs with values loaded from memory.

WRMSRLIST takes three implied input operands:

- RSI: Linear address of a table of MSR addresses (8 bytes per address)[1].
- RDI: Linear address of a table from which MSR data is loaded (8 bytes per MSR).
- RCX: 64-bit bitmask of valid bits for the MSRs. Bit 0 is the valid bit for entry 0 in each table, etc.

For each RCX bit [n] from 0 to 63, if RCX[n] is 1, WRMSRLIST will write the MSR specified at entry [n] in the RSI-based table with the value read from memory at the entry [n] in the RDI-based table.

This implies a maximum of 64 MSRs that can be processed by this instruction. The processor will clear RCX[n] after it finishes handling that MSR. Similar to repeated string operations, WRMSRLIST supports partial completion for interrupts, exceptions, and traps. In these situations, the RIP register saved will point to the MSRLIST instruction while the RCX register will have cleared bits corresponding to all completed iterations.

This instruction must be executed at privilege level 0; otherwise, a general protection exception #GP(0) is generated. This instruction performs MSR-specific checks in the same manner as WRMSR.

Like WRMSRNS (and unlike WRMSR), WRMSRLIST is not defined as a serializing instruction (see "Serializing Instructions" in Chapter 11 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A). This means that software should not rely on WRMSRLIST to drain all buffered writes to memory before the next instruction is fetched and executed. For implementation reasons, some processors may serialize when writing certain MSRs, even though that is not guaranteed.

Like WRMSR and WRMSRNS, WRMSRLIST ensures that all operations before WRMSRLIST do not use any new MSR value and that all operations after WRMSRLIST do use the new values. An exception to this rule is certain store related performance-monitor events that only count stores when they are drained to memory. Since WRMSRLIST is not a serializing instruction, if software uses WRMSRLIST to change the controls for such performance-monitor events, stores issued before WRMSRLIST may be counted based on the controls established by WRMSRLIST. Software can insert the SERIALIZE instruction before the WRMSRLIST if so desired.

Those MSRs that cause a TLB invalidation when they are written via WRMSR (e.g., MTRRs) will also cause the same TLB invalidation when written by WRMSRLIST.

In places where WRMSR is being used as a proxy for a serializing instruction, a different serializing instruction can be used (e.g., SERIALIZE).

WRMSRLIST writes MSRs in order, which means the processor will ensure that an MSR in iteration "n" will be written only after previous iterations ("n-1"). If the older MSR writes had a side effect that affects the behavior of the next MSR, the processor will ensure that side effect is honored.

The processor is allowed (but not required) to "load ahead" in the list. The following are examples of things the processor may do:

- Use an old memory type or TLB entry for loads or stores to memory containing the tables despite an MSR written by a previous iteration changing MTRR or invalidating TLBs.

1. Since MSR addresses are only 32-bits wide, bits 63:32 of each MSR address table entry is reserved.

- Cause a page fault for access to a table entry after the n^{th}, despite the processor having written only n MSRs.[1]

Operation

```
DO WHILE RCX != 0
    MSR_index := position of least significant bit set in RCX;
    Load MSR_address_table_entry from 8 bytes at the linear address RSI + (MSR_index * 8);
    IF MSR_address_table_entry[63:32] != 0 THEN #GP(0); FI;
    MSR_address := MSR_address_table_entry[31:0];
    Load MSR_data from 8 bytes at the linear address RDI + (MSR_index * 8);
    IF WRMSR of MSR_data to the MSR with address MSR_address would #GP THEN #GP(0); FI;
    Load the MSR with address MSR_address with MSR_data;
    RCX[MSR_index] := 0;
    Allow delivery of any pending interrupts or traps;
OD;
```

Flags Affected

None.

Protected Mode Exceptions

#UD	The WRMSRLIST instruction is not recognized in protected mode.

Real-Address Mode Exceptions

#UD	The WRMSRLIST instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The WRMSRLIST instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	The WRMSRLIST instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If RSI [2:0] ≠ 0, RDI [2:0] ≠ 0, or bits 63:32 of an MSR-address table entry are not all zero.
	If an execution of WRMSR to a specified MSR with a specified value would generate a general-protection exception (#GP(0)).
	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.
	If CPUID.07H.01H:EAX.MSRLIST[27] = 0.

1. For example, the processor may take a page fault due to a linear address for the 10th entry in the MSR address table despite only having completed the MSR writes up to entry 5.

WRMSRNS—Non-Serializing Write to Model Specific Register

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 C6 WRMSRNS	ZO	V/V	WRMSRNS	Write the value in EDX:EAX to MSR specified by ECX.
VEX.128.F3.MAP7:W0.F6 11:000:bbb WRMSRNS imm32, r64	IM	V/N.E.	MSR_IMM	Write the value of register bbb to the MSR specified in the 32-bit immediate.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A
IM	imm32	ModRM:r/m (r)	N/A	N/A

Description

WRMSRNS is an instruction that behaves like WRMSR except that it is not a serializing instruction by default. It can be executed only at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. WRMSRNS has an **implicit** form and an **immediate** form.

The implicit form writes the contents of registers EDX:EAX into the 64-bit model specific register (MSR) specified in the ECX register. The contents of the EDX register are copied to the high-order 32 bits of the selected MSR and the contents of the EAX register are copied to the low-order 32 bits of the MSR. The high-order 32 bits of RAX, RCX, and RDX are ignored.

The immediate form writes the contents of operand 2 into the 64-bit MSR specified by operand 1. Operand 2 is a general-purpose register, while operand 1 is an immediate. The immediate form can be used only in 64-bit mode; otherwise, an invalid-opcode exception (#UD) will be generated.

Unlike WRMSR, WRMSRNS is not defined as a serializing instruction (see Section 11.7.10, "Serializing Instructions" of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A). This means that software should not rely on it to drain all buffered writes to memory before the next instruction is fetched and executed. For implementation reasons, some processors may serialize when writing certain MSRs, even though that is not guaranteed.

Like WRMSR, WRMSRNS will ensure that all operations before it do not use the new MSR value and that all operations after the WRMSRNS do use the new value. An exception to this rule is certain store related performance-monitor events that only count stores when they are drained to memory. Since WRMSRNS is not a serializing instruction, if software uses WRMSRNS to change the controls for such performance-monitor events, stores issued before WRMSRMS may be counted based on the controls established by WRMSRNS. Software can insert the SERIALIZE instruction before the WRMSRNS if so desired.

Those MSRs that cause a TLB invalidation when they are written via WRMSR (e.g., MTRRs) will also cause the same TLB invalidation when written by WRMSRNS.

In order to improve performance, software may replace WRMSR with WRMSRNS. In places where WRMSR is being used as a proxy for a serializing instruction, a different serializing instruction can be used (e.g., SERIALIZE).

Operation

```
IF implicit form
    THEN
        MSR[ECX] := EDX:EAX;
    ELSE (* immediate form *)
        MSR[DEST] := SRC;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If the specified MSR address is reserved or unimplemented MSR.
	If the source data sets bits that are reserved in the specified MSR.
	If the source data contains a non-canonical address and the specified MSR is one of the following: IA32_BNDCFGS, IA32_DS_AREA, IA32_FS_BASE, IA32_GS_BASE, IA32_INTERRUPT_SSP_TABLE_ADDR, IA32_KERNEL_GS_BASE, IA32_LSTAR, IA32_PL0_SSP, IA32_PL1_SSP, IA32_PL2_SSP, IA32_PL3_SSP, IA32_RTIT_ADDR0_A, IA32_RTIT_ADDR0_B, IA32_RTIT_ADDR1_A, IA32_RTIT_ADDR1_B, IA32_RTIT_ADDR2_A, IA32_RTIT_ADDR2_B, IA32_RTIT_ADDR3_A, IA32_RTIT_ADDR3_B, IA32_S_CET, IA32_SYSENTER_EIP, IA32_SYSENTER_ESP, IA32_UINTR_HANDLER, IA32_UINTR_PD, IA32_UINTR_STACKADJUST, IA32_U_CET, and IA32_UINTR_TT.
#UD	If the LOCK prefix is used.
	If the immediate form is used.

Real-Address Mode Exceptions

#GP(0)	If the specified MSR address is reserved or unimplemented MSR.
	If the source data sets bits that are reserved in the specified MSR.
	If the source data contains a non-canonical address and the specified MSR is one of the following: IA32_BNDCFGS, IA32_DS_AREA, IA32_FS_BASE, IA32_GS_BASE, IA32_INTERRUPT_SSP_TABLE_ADDR, IA32_KERNEL_GS_BASE, IA32_LSTAR, IA32_PL0_SSP, IA32_PL1_SSP, IA32_PL2_SSP, IA32_PL3_SSP, IA32_RTIT_ADDR0_A, IA32_RTIT_ADDR0_B, IA32_RTIT_ADDR1_A, IA32_RTIT_ADDR1_B, IA32_RTIT_ADDR2_A, IA32_RTIT_ADDR2_B, IA32_RTIT_ADDR3_A, IA32_RTIT_ADDR3_B, IA32_S_CET, IA32_SYSENTER_EIP, IA32_SYSENTER_ESP, IA32_UINTR_HANDLER, IA32_UINTR_PD, IA32_UINTR_STACKADJUST, IA32_U_CET, and IA32_UINTR_TT.
#UD	If the LOCK prefix is used.
	If the immediate form is used.

Virtual-8086 Mode Exceptions

#GP(0)	For the implicit form, the WRMSRNS instruction is not recognized in virtual-8086 mode.
#UD	For the immediate form, the WRMSRNS 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 specified MSR address is reserved or unimplemented MSR.
	If the source data sets bits that are reserved in the specified MSR.
	If the source data contains a non-canonical address and the specified MSR is one of the following: IA32_BNDCFGS, IA32_DS_AREA, IA32_FS_BASE, IA32_GS_BASE, IA32_INTERRUPT_SSP_TABLE_ADDR, IA32_KERNEL_GS_BASE, IA32_LSTAR, IA32_PL0_SSP, IA32_PL1_SSP, IA32_PL2_SSP, IA32_PL3_SSP, IA32_RTIT_ADDR0_A, IA32_RTIT_ADDR0_B, IA32_RTIT_ADDR1_A, IA32_RTIT_ADDR1_B, IA32_RTIT_ADDR2_A, IA32_RTIT_ADDR2_B, IA32_RTIT_ADDR3_A, IA32_RTIT_ADDR3_B, IA32_S_CET, IA32_SYSENTER_EIP, IA32_SYSENTER_ESP, IA32_UINTR_HANDLER, IA32_UINTR_PD, IA32_UINTR_STACKADJUST, IA32_U_CET, and IA32_UINTR_TT.

#UD	If the LOCK prefix is used. If the immediate form is used and CPUID.07H.01H:ECX.MSR_IMM[5] = 0.

WRPKRU—Write Data to User Page Key Register

Opcode/ Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 EF WRPKRU	ZO	V/V	OSPKE	Writes EAX into PKRU.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Writes the value of EAX into PKRU. ECX and EDX must be 0 when WRPKRU is executed; otherwise, a general-protection exception (#GP) occurs.

WRPKRU can be executed only if CR4.PKE = 1; otherwise, an invalid-opcode exception (#UD) occurs. Software can discover the value of CR4.PKE by examining CPUID.07H.00H:ECX.OSPKE[4].

On processors that support the Intel 64 Architecture, the high-order 32-bits of RCX, RDX, and RAX are ignored.

WRPKRU will never execute speculatively. Memory accesses affected by PKRU register will not execute (even speculatively) until all prior executions of WRPKRU have completed execution and updated the PKRU register.

Operation

```
IF (ECX = 0 AND EDX = 0)
    THEN PKRU := EAX;
    ELSE #GP(0);
FI;
```

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

WRPKRU void _wrpkru(uint32_t);

Protected Mode Exceptions

#GP(0)	If ECX ≠ 0.
	If EDX ≠ 0.
#UD	If the LOCK prefix is used.
	If CR4.PKE = 0.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

WRSSD/WRSSQ—Write to Shadow Stack

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 F6 !(11):rrr:bbb WRSSD m32, r32	MR	V/V	CET_SS	Write 4 bytes to shadow stack.
REX.W 0F 38 F6 !(11):rrr:bbb WRSSQ m64, r64	MR	V/N.E.	CET_SS	Write 8 bytes to shadow stack.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Writes bytes in register source to the shadow stack.

Operation

```
IF CPL = 3
    IF (CR4.CET & IA32_U_CET.SH_STK_EN) = 0
        THEN #UD; FI;
    IF (IA32_U_CET.WR_SHSTK_EN) = 0
        THEN #UD; FI;
ELSE
    IF (CR4.CET & IA32_S_CET.SH_STK_EN) = 0
        THEN #UD; FI;
    IF (IA32_S_CET.WR_SHSTK_EN) = 0
        THEN #UD; FI;
FI;
DEST_LA = Linear_Address(mem operand)
IF (operand size is 64 bit)
    THEN
        (* Destination not 8B aligned *)
        IF DEST_LA[2:0]
            THEN GP(0); FI;
        Shadow_stack_store 8 bytes of SRC to DEST_LA;
    ELSE
        (* Destination not 4B aligned *)
        IF DEST_LA[1:0]
            THEN GP(0); FI;
        Shadow_stack_store 4 bytes of SRC[31:0] to DEST_LA;
FI;
```

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

WRSSD void _wrssd(__int32, void *);
WRSSQ void _wrssq(__int64, void *);

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	If CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
	If CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
	If CPL = 3 and IA32_U_CET.WR_SHSTK_EN = 0.
	If CPL < 3 and IA32_S_CET.WR_SHSTK_EN = 0.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If destination is located in a non-writeable segment.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If linear address of destination is not 4 byte aligned.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs if destination is not a user shadow stack when CPL3 and not a supervisor shadow stack when CPL < 3.
	Other terminal and non-terminal faults.

Real-Address Mode Exceptions

#UD	The WRSS instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The WRSS instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	If CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
	If CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
	If CPL = 3 and IA32_U_CET.WR_SHSTK_EN = 0.
	If CPL < 3 and IA32_S_CET.WR_SHSTK_EN = 0.
#PF(fault-code)	If a page fault occurs if destination is not a user shadow stack when CPL3 and not a supervisor shadow stack when CPL < 3.
	Other terminal and non-terminal faults.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	If CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
	If CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
	If CPL = 3 and IA32_U_CET.WR_SHSTK_EN = 0.
	If CPL < 3 and IA32_S_CET.WR_SHSTK_EN = 0.
#GP(0)	If a memory address is in a non-canonical form.
	If linear address of destination is not 4 byte aligned.
#PF(fault-code)	If a page fault occurs if destination is not a user shadow stack when CPL3 and not a supervisor shadow stack when CPL < 3.
	Other terminal and non-terminal faults.

WRUSSD/WRUSSQ—Write to User Shadow Stack

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 F5 !(11):rrr:bbb WRUSSD m32, r32	MR	V/V	CET_SS	Write 4 bytes to shadow stack.
66 REX.W 0F 38 F5 !(11):rrr:bbb WRUSSQ m64, r64	MR	V/N.E.	CET_SS	Write 8 bytes to shadow stack.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Writes bytes in register source to a user shadow stack pag.

Operation

```
IF CR4.CET = 0
    THEN #UD; FI;
IF CPL > 0
    THEN #GP(0); FI;
DEST_LA = Linear_Address(mem operand)
IF (operand size is 64 bit)
    THEN
        (* Destination not 8B aligned *)
        IF DEST_LA[2:0]
            THEN GP(0); FI;
        Shadow_stack_store 8 bytes of SRC to DEST_LA as user-mode access;
    ELSE
        (* Destination not 4B aligned *)
        IF DEST_LA[1:0]
            THEN GP(0); FI;
        Shadow_stack_store 4 bytes of SRC[31:0] to DEST_LA as user-mode access;
FI;
```

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

WRUSSD void _wrussd(__int32, void *);
WRUSSQ void _wrussq(__int64, void *);

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If destination is located in a non-writeable segment.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If linear address of destination is not 4 byte aligned.
	If CPL is not 0.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If destination is not a user shadow stack.
	Other terminal and non-terminal faults.

Real-Address Mode Exceptions

#UD	The WRUSS instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The WRUSS instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
#GP(0)	If a memory address is in a non-canonical form.
	If linear address of destination is not 4 byte aligned.
	If CPL is not 0.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If destination is not a user shadow stack.
	Other terminal and non-terminal faults.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
#GP(0)	If a memory address is in a non-canonical form.
	If linear address of destination is not 4 byte aligned.
	If CPL is not 0.
#PF(fault-code)	If destination is not a user shadow stack.
	Other terminal and non-terminal faults.

XABORT—Transactional Abort

Opcode/Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
C6 F8 ib XABORT imm8	A	V/V	RTM	Causes an RTM abort if in RTM execution.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	imm8	N/A	N/A	N/A

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;
    SUSLDTRK_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	If CPUID.07H.00H:EBX.RTM[11] = 0. If LOCK prefix is used.

XACQUIRE/XRELEASE—Hardware Lock Elision Prefix Hints

Opcode/Instruction	64/32bit Mode Support	CPUID Feature Flag	Description
F2 XACQUIRE	V/V	HLE[1]	A hint used with an "XACQUIRE-enabled" instruction to start lock elision on the instruction memory operand address.
F3 XRELEASE	V/V	HLE	A hint used with an "XRELEASE-enabled" instruction to end lock elision on the instruction memory operand address.

NOTES:

1. Software is not required to check the HLE feature flag to use XACQUIRE or XRELEASE, as they are treated as regular prefix if HLE feature flag reports 0.

Description

The XACQUIRE prefix is a hint to start lock elision on the memory address specified by the instruction and the XRELEASE prefix is a hint to end lock elision on the memory address specified by the instruction.

The XACQUIRE prefix hint can only be used with the following instructions (these instructions are also referred to as XACQUIRE-enabled when used with the XACQUIRE prefix):

- Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.
- The XCHG instruction either with or without the presence of the LOCK prefix.

The XRELEASE prefix hint can only be used with the following instructions (also referred to as XRELEASE-enabled when used with the XRELEASE prefix):

- Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.
- The XCHG instruction either with or without the presence of the LOCK prefix.
- The "MOV mem, reg" (Opcode 88H/89H) and "MOV mem, imm" (Opcode C6H/C7H) instructions. In these cases, the XRELEASE is recognized without the presence of the LOCK prefix.

The lock variables must satisfy the guidelines described in Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, Section 16.3.3, for elision to be successful, otherwise an HLE abort may be signaled.

If an encoded byte sequence that meets XACQUIRE/XRELEASE requirements includes both prefixes, then the HLE semantic is determined by the prefix byte that is placed closest to the instruction opcode. For example, an F3F2C6 will not be treated as a XRELEASE-enabled instruction since the F2H (XACQUIRE) is closest to the instruction opcode C6. Similarly, an F2F3F0 prefixed instruction will be treated as a XRELEASE-enabled instruction since F3H (XRELEASE) is closest to the instruction opcode.

Intel 64 and IA-32 Compatibility

The effect of the XACQUIRE/XRELEASE prefix hint is the same in non-64-bit modes and in 64-bit mode.

For instructions that do not support the XACQUIRE hint, the presence of the F2H prefix behaves the same way as prior hardware, according to

- REPNE/REPNZ semantics for string instructions,
- Serve as SIMD prefix for legacy SIMD instructions operating on XMM register
- Cause #UD if prepending the VEX prefix.
- Undefined for non-string instructions or other situations.

For instructions that do not support the XRELEASE hint, the presence of the F3H prefix behaves the same way as in prior hardware, according to

- REP/REPE/REPZ semantics for string instructions,
- Serve as SIMD prefix for legacy SIMD instructions operating on XMM register
- Cause #UD if prepending the VEX prefix.
- Undefined for non-string instructions or other situations.

Operation

XACQUIRE

```
IF XACQUIRE-enabled instruction
    THEN
        IF (HLE_NEST_COUNT < MAX_HLE_NEST_COUNT) THEN
            HLE_NEST_COUNT++
            IF (HLE_NEST_COUNT = 1) THEN
                HLE_ACTIVE := 1
                IF 64-bit mode
                    THEN
                        restartRIP := instruction pointer of the XACQUIRE-enabled instruction
                    ELSE
                        restartEIP := instruction pointer of the XACQUIRE-enabled instruction
                FI;
                Enter HLE Execution (* record register state, start tracking memory state *)
            FI; (* HLE_NEST_COUNT = 1*)
            IF ElisionBufferAvailable
                THEN
                    Allocate elision buffer
                    Record address and data for forwarding and commit checking
                    Perform elision
                ELSE
                    Perform lock acquire operation transactionally but without elision
            FI;
        ELSE (* HLE_NEST_COUNT = MAX_HLE_NEST_COUNT*)
                GOTO HLE_ABORT_PROCESSING
        FI;
    ELSE
        Treat instruction as non-XACQUIRE F2H prefixed legacy instruction
FI;
```

XRELEASE

```
IF XRELEASE-enabled instruction
    THEN
        IF (HLE_NEST_COUNT > 0)
            THEN
                HLE_NEST_COUNT--
                IF lock address matches in elision buffer THEN
                    IF lock satisfies address and value requirements THEN
                        Deallocate elision buffer
                    ELSE
                        GOTO HLE_ABORT_PROCESSING
                    FI;
                FI;
                IF (HLE_NEST_COUNT = 0)
                    THEN
                        IF NoAllocatedElisionBuffer
                            THEN
                                Try to commit transactional execution
                                IF fail to commit transactional execution
                                    THEN
                                        GOTO HLE_ABORT_PROCESSING;
                                    ELSE (* commit success *)
                                        HLE_ACTIVE := 0
                                FI;
                            ELSE
                                GOTO HLE_ABORT_PROCESSING
                        FI;
                FI;
        FI; (* HLE_NEST_COUNT > 0 *)
    ELSE
        Treat instruction as non-XRELEASE F3H prefixed legacy instruction
FI;

(* For any HLE abort condition encountered during HLE execution *)
HLE_ABORT_PROCESSING:
    HLE_ACTIVE := 0
    HLE_NEST_COUNT := 0
    Restore architectural register state
    Discard memory updates performed in transaction
    Free any allocated lock elision buffers
    IF 64-bit mode
        THEN
            RIP := restartRIP
        ELSE
            EIP := restartEIP
    FI;
    Execute and retire instruction at RIP (or EIP) and ignore any HLE hint
END
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

#GP(0)	If the use of prefix causes instruction length to exceed 15 bytes.

XADD—Exchange and Add

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F C0 /r	XADD r/m8[1], r8[1]	MR	Valid	Valid	Exchange r8 and r/m8; load sum into r/m8.
0F C1 /r	XADD r/m16, r16	MR	Valid	Valid	Exchange r16 and r/m16; load sum into r/m16.
0F C1 /r	XADD r/m32, r32	MR	Valid	Valid	Exchange r32 and r/m32; load sum into r/m32.
REX.W + 0F C1 /r	XADD r/m64, r64	MR	Valid	N.E.	Exchange r64 and r/m64; load sum into r/m64.

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r, w)	N/A	N/A

Description

Exchanges the first operand (destination operand) with the second operand (source operand), then loads the sum of the two values into the destination operand. The destination operand can be a register or a memory location; the source operand is a register.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

IA-32 Architecture Compatibility

IA-32 processors earlier than the Intel486 processor do not recognize this instruction. If this instruction is used, you should provide an equivalent code sequence that runs on earlier processors.

Operation

```
TEMP := SRC + DEST;
SRC := DEST;
DEST := TEMP;
```

Flags Affected

The CF, PF, AF, SF, ZF, and OF flags are set according to the result of the addition, which is stored in the destination operand.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

XBEGIN—Transactional Begin

Opcode/Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
C7 F8 XBEGIN rel16	A	V/V	RTM	Specifies the start of an RTM region. Provides a 16-bit relative offset to compute the address of the fallback instruction address at which execution resumes following an RTM abort.
C7 F8 XBEGIN rel32	A	V/V	RTM	Specifies the start of an RTM region. Provides a 32-bit relative offset to compute the address of the fallback instruction address at which execution resumes following an RTM abort.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	Offset	N/A	N/A	N/A

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. (Use of the 16-bit operand size does not cause this address to be truncated to 16 bits, unlike a near jump to a relative offset.)

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.

Execution of XBEGIN while in a suspend read address tracking region causes a transactional abort.

Operation

XBEGIN

```
IF RTM_NEST_COUNT < MAX_RTM_NEST_COUNT AND SUSLDTRK_ACTIVE = 0
    THEN
        RTM_NEST_COUNT++
        IF RTM_NEST_COUNT = 1 THEN
            IF 64-bit Mode
                THEN
                    IF OperandSize = 16
                        THEN fallbackRIP := RIP + SignExtend64(rel16);
                        ELSE fallbackRIP := RIP + SignExtend64(rel32);
                    FI;
                    IF fallbackRIP is not canonical
                        THEN #GP(0);
                    FI;
                ELSE
                    IF OperandSize = 16
                        THEN fallbackEIP := EIP + SignExtend32(rel16);
                        ELSE fallbackEIP := EIP + rel32;
                    FI;
                    IF fallbackEIP outside code segment limit
                        THEN #GP(0);
                    FI;
            FI;
```

```
            RTM_ACTIVE := 1
            Enter RTM Execution (* record register state, start tracking memory state*)
        FI; (* RTM_NEST_COUNT = 1 *)
    ELSE (* RTM_NEST_COUNT = MAX_RTM_NEST_COUNT OR SUSLDTRK_ACTIVE = 1 *)
        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
    SUSLDTRK_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	If CPUID.07H.00H:EBX.RTM[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	If CPUID.07H.00H:EBX.RTM[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	If CPUID.07H.00H:EBX.RTM[11] = 0.
	If LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-bit Mode Exceptions

#UD	If CPUID.07H.00H:EBX.RTM[11] = 0.
	If LOCK prefix is used.
#GP(0)	If the fallback address is non-canonical.

XCHG—Exchange Register/Memory With Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
90+rw	XCHG AX, r16	O	Valid	Valid	Exchange r16 with AX.
90+rw	XCHG r16, AX	O	Valid	Valid	Exchange AX with r16.
90+rd	XCHG EAX, r32	O	Valid	Valid	Exchange r32 with EAX.
REX.W + 90+rd	XCHG RAX, r64	O	Valid	N.E.	Exchange r64 with RAX.
90+rd	XCHG r32, EAX	O	Valid	Valid	Exchange EAX with r32.
REX.W + 90+rd	XCHG r64, RAX	O	Valid	N.E.	Exchange RAX with r64.
86 /r	XCHG r/m8[1], r8[1]	MR	Valid	Valid	Exchange r8 (byte register) with byte from r/m8.
86 /r	XCHG r8[1], r/m8[1]	RM	Valid	Valid	Exchange byte from r/m8 with r8 (byte register).
87 /r	XCHG r/m16, r16	MR	Valid	Valid	Exchange r16 with word from r/m16.
87 /r	XCHG r16, r/m16	RM	Valid	Valid	Exchange word from r/m16 with r16.
87 /r	XCHG r/m32, r32	MR	Valid	Valid	Exchange r32 with doubleword from r/m32.
REX.W + 87 /r	XCHG r/m64, r64	MR	Valid	N.E.	Exchange r64 with quadword from r/m64.
87 /r	XCHG r32, r/m32	RM	Valid	Valid	Exchange doubleword from r/m32 with r32.
REX.W + 87 /r	XCHG r64, r/m64	RM	Valid	N.E.	Exchange quadword from r/m64 with r64.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
O	AX/EAX/RAX (r, w)	opcode + rd (r, w)	N/A	N/A
O	opcode + rd (r, w)	AX/EAX/RAX (r, w)	N/A	N/A
MR	ModRM:r/m (r, w)	ModRM:reg (r)	N/A	N/A
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Exchanges the contents of the destination (first) and source (second) operands. The operands can be two general-purpose registers or a register and a memory location. If a memory operand is referenced, the processor's locking protocol is automatically implemented for the duration of the exchange operation, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL. (See the LOCK prefix description in this chapter for more information on the locking protocol.)

This instruction is useful for implementing semaphores or similar data structures for process synchronization. (See "Bus Locking" in Chapter 9 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, for more information on bus locking.)

The XCHG instruction can also be used instead of the BSWAP instruction for 16-bit operands.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

NOTE

XCHG (E)AX, (E)AX (encoded instruction byte is 90H) is an alias for NOP regardless of data size prefixes, including REX.W.

Operation

```
TEMP := DEST;
DEST := SRC;
SRC := TEMP;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If either operand is in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

XEND—Transactional End

Opcode/Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
NP 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	N/A	N/A	N/A	N/A

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.

Execution of XEND outside a transactional region causes a general-protection exception (#GP). Execution of XEND while in a suspend read address tracking region causes a transactional abort.

Operation

XEND

```
IF (RTM_ACTIVE = 0) THEN
    SIGNAL #GP
ELSE
    IF SUSLDTRK_ACTIVE = 1
        THEN GOTO RTM_ABORT_PROCESSING;
    FI;
    RTM_NEST_COUNT--
    IF (RTM_NEST_COUNT = 0) THEN
        Try to commit transaction
        IF fail to commit transactional execution
            THEN
                GOTO RTM_ABORT_PROCESSING;
            ELSE (* commit success *)
                RTM_ACTIVE := 0
        FI;
    FI;
FI;

(* For any RTM abort condition encountered during RTM execution *)
RTM_ABORT_PROCESSING:
    Restore architectural register state
    Discard memory updates performed in transaction
    Update EAX with status
    RTM_NEST_COUNT := 0
    RTM_ACTIVE := 0
    SUSLDTRK_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	If CPUID.07H.00H:EBX.RTM[11] = 0.
	If LOCK prefix is used.
#GP(0)	If RTM_ACTIVE = 0.

XGETBV—Get Value of Extended Control Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F 01 D0	XGETBV	ZO	Valid	Valid	Reads an XCR specified by ECX into EDX:EAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Reads the contents of the extended control register (XCR) specified in the ECX register into registers EDX:EAX. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The EDX register is loaded with the high-order 32 bits of the XCR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.) If fewer than 64 bits are implemented in the XCR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

XCR0 is supported on any processor that supports the XGETBV instruction. If CPUID.0DH.01H:EAX.XGETBV1[2] = 1, executing XGETBV with ECX = 1 returns in EDX:EAX the logical-AND of XCR0 and the current value of the XINUSE state-component bitmap. This allows software to discover the state of the init optimization used by XSAVEOPT and XSAVES. See Chapter 13, "Managing State Using the XSAVE Feature Set," in Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Use of any other value for ECX results in a general-protection (#GP) exception.

Operation

```
EDX:EAX := XCR[ECX];
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XGETBV unsigned __int64 _xgetbv(unsigned int);

Protected Mode Exceptions

#GP(0)	If an invalid XCR is specified in ECX (includes ECX = 1 if CPUID.0DH.01H:EAX.XGETBV1[2] = 0).
#UD	If CPUID.01H:ECX.XSAVE[26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP(0)	If an invalid XCR is specified in ECX (includes ECX = 1 if CPUID.0DH.01H:EAX.XGETBV1[2] = 0).
#UD	If CPUID.01H:ECX.XSAVE[26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

XLAT/XLATB—Table Look-up Translation

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
D7	XLAT m8	ZO	Valid	Valid	Set AL to memory byte DS:[(E)BX + unsigned AL].
D7	XLATB	ZO	Valid	Valid	Set AL to memory byte DS:[(E)BX + unsigned AL].
REX.W + D7	XLATB	ZO	Valid	N.E.	Set AL to memory byte [RBX + unsigned AL].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Locates a byte entry in a table in memory, using the contents of the AL register as a table index, then copies the contents of the table entry back into the AL register. The index in the AL register is treated as an unsigned integer. The XLAT and XLATB instructions get the base address of the table in memory from either the DS:EBX or the DS:BX registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). (The DS segment may be overridden with a segment override prefix.)

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operand" form and the "no-operand" form. The explicit-operand form (specified with the XLAT mnemonic) allows the base address of the table to be specified explicitly with a symbol. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the symbol does not have to specify the correct base address. The base address is always specified by the DS:(E)BX registers, which must be loaded correctly before the XLAT instruction is executed.

The no-operands form (XLATB) provides a "short form" of the XLAT instructions. Here also the processor assumes that the DS:(E)BX registers contain the base address of the table.

In 64-bit mode, operation is similar to that in legacy or compatibility mode. AL is used to specify the table index (the operand size is fixed at 8 bits). RBX, however, is used to specify the table's base address. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF AddressSize = 16
    THEN
        AL := (DS:BX + ZeroExtend(AL));
    ELSE IF (AddressSize = 32)
        AL := (DS:EBX + ZeroExtend(AL)); FI;
    ELSE (AddressSize = 64)
        AL := (RBX + ZeroExtend(AL));
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

XOR—Logical Exclusive OR

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
34 ib	XOR AL, imm8	I	Valid	Valid	AL XOR imm8.
35 iw	XOR AX, imm16	I	Valid	Valid	AX XOR imm16.
35 id	XOR EAX, imm32	I	Valid	Valid	EAX XOR imm32.
REX.W + 35 id	XOR RAX, imm32	I	Valid	N.E.	RAX XOR imm32 (sign-extended).
80 /6 ib	XOR r/m8[1], imm8	MI	Valid	Valid	r/m8 XOR imm8.
81 /6 iw	XOR r/m16, imm16	MI	Valid	Valid	r/m16 XOR imm16.
81 /6 id	XOR r/m32, imm32	MI	Valid	Valid	r/m32 XOR imm32.
REX.W + 81 /6 id	XOR r/m64, imm32	MI	Valid	N.E.	r/m64 XOR imm32 (sign-extended).
83 /6 ib	XOR r/m16, imm8	MI	Valid	Valid	r/m16 XOR imm8 (sign-extended).
83 /6 ib	XOR r/m32, imm8	MI	Valid	Valid	r/m32 XOR imm8 (sign-extended).
REX.W + 83 /6 ib	XOR r/m64, imm8	MI	Valid	N.E.	r/m64 XOR imm8 (sign-extended).
30 /r	XOR r/m8[1], r8[1]	MR	Valid	Valid	r/m8 XOR r8.
31 /r	XOR r/m16, r16	MR	Valid	Valid	r/m16 XOR r16.
31 /r	XOR r/m32, r32	MR	Valid	Valid	r/m32 XOR r32.
REX.W + 31 /r	XOR r/m64, r64	MR	Valid	N.E.	r/m64 XOR r64.
32 /r	XOR r8,[1] r/m8[1]	RM	Valid	Valid	r8 XOR r/m8.
33 /r	XOR r16, r/m16	RM	Valid	Valid	r16 XOR r/m16.
33 /r	XOR r32, r/m32	RM	Valid	Valid	r32 XOR r/m32.
REX.W + 33 /r	XOR r64, r/m64	RM	Valid	N.E.	r64 XOR r/m64.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	imm8/16/32	N/A	N/A
MI	ModRM:r/m (r, w)	imm8/16/32	N/A	N/A
MR	ModRM:r/m (r, w)	ModRM:reg (r)	N/A	N/A
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A

Description

Performs a bitwise exclusive OR (XOR) operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is 1 if the corresponding bits of the operands are different; each bit is 0 if the corresponding bits are the same.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST := DEST XOR SRC;
```

Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

XORPD—Bitwise Logical XOR of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 57/r XORPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical XOR of packed double precision floating-point values in xmm1 and xmm2/mem.
VEX.128.66.0F.WIG 57 /r VXORPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical XOR of packed double precision floating-point values in xmm2 and xmm3/mem.
VEX.256.66.0F.WIG 57 /r VXORPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical XOR of packed double precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.66.0F.W1 57 /r VXORPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical XOR of packed double precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.256.66.0F.W1 57 /r VXORPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical XOR of packed double precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.512.66.0F.W1 57 /r VXORPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ OR AVX10.1	Return the bitwise logical XOR of packed double precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical XOR of the two, four or eight packed double precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand can be a ZMM register or a vector memory location. The destination operand is a ZMM register conditionally updated with write-mask k1.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VXORPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := SRC1[i+63:i] BITWISE XOR SRC2[63:0];
                ELSE DEST[i+63:i] := SRC1[i+63:i] BITWISE XOR SRC2[i+63:i];
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+63:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VXORPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0] BITWISE XOR SRC2[63:0]
DEST[127:64] := SRC1[127:64] BITWISE XOR SRC2[127:64]
DEST[191:128] := SRC1[191:128] BITWISE XOR SRC2[191:128]
DEST[255:192] := SRC1[255:192] BITWISE XOR SRC2[255:192]
DEST[MAXVL-1:256] := 0
```

VXORPD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] BITWISE XOR SRC2[63:0]
DEST[127:64] := SRC1[127:64] BITWISE XOR SRC2[127:64]
DEST[MAXVL-1:128] := 0
```

XORPD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] BITWISE XOR SRC[63:0]
DEST[127:64] := DEST[127:64] BITWISE XOR SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VXORPD __m512d _mm512_xor_pd (__m512d a, __m512d b);
VXORPD __m512d _mm512_mask_xor_pd (__m512d a, __mmask8 m, __m512d b);
VXORPD __m512d _mm512_maskz_xor_pd (__mmask8 m, __m512d a);
VXORPD __m256d _mm256_xor_pd (__m256d a, __m256d b);
VXORPD __m256d _mm256_mask_xor_pd (__m256d a, __mmask8 m, __m256d b);
VXORPD __m256d _mm256_maskz_xor_pd (__mmask8 m, __m256d a);
XORPD __m128d _mm_xor_pd (__m128d a, __m128d b);
VXORPD __m128d _mm_mask_xor_pd (__m128d a, __mmask8 m, __m128d b);
VXORPD __m128d _mm_maskz_xor_pd (__mmask8 m, __m128d a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instructions, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-49, "Type E4 Class Exception Conditions."

XORPS—Bitwise Logical XOR of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 57 /r XORPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical XOR of packed single precision floating-point values in xmm1 and xmm2/mem.
VEX.128.0F.WIG 57 /r VXORPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical XOR of packed single precision floating-point values in xmm2 and xmm3/mem.
VEX.256.0F.WIG 57 /r VXORPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical XOR of packed single precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.0F.W0 57 /r VXORPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical XOR of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.256.0F.W0 57 /r VXORPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical XOR of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.512.0F.W0 57 /r VXORPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512DQ OR AVX10.1	Return the bitwise logical XOR of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical XOR of the four, eight or sixteen packed single precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand can be a ZMM register or a vector memory location. The destination operand is a ZMM register conditionally updated with write-mask k1.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VXORPS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := SRC1[i+31:i] BITWISE XOR SRC2[31:0];
                ELSE DEST[i+31:i] := SRC1[i+31:i] BITWISE XOR SRC2[i+31:i];
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+31:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VXORPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[63:32] := SRC1[63:32] BITWISE XOR SRC2[63:32]
DEST[95:64] := SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[127:96] := SRC1[127:96] BITWISE XOR SRC2[127:96]
DEST[159:128] := SRC1[159:128] BITWISE XOR SRC2[159:128]
DEST[191:160] := SRC1[191:160] BITWISE XOR SRC2[191:160]
DEST[223:192] := SRC1[223:192] BITWISE XOR SRC2[223:192]
DEST[255:224] := SRC1[255:224] BITWISE XOR SRC2[255:224].
DEST[MAXVL-1:256] := 0
```

VXORPS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[63:32] := SRC1[63:32] BITWISE XOR SRC2[63:32]
DEST[95:64] := SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[127:96] := SRC1[127:96] BITWISE XOR SRC2[127:96]
DEST[MAXVL-1:128] := 0
```

XORPS (128-bit Legacy SSE Version)

```
DEST[31:0] := SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[63:32] := SRC1[63:32] BITWISE XOR SRC2[63:32]
DEST[95:64] := SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[127:96] := SRC1[127:96] BITWISE XOR SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VXORPS __m512 _mm512_xor_ps (__m512 a, __m512 b);
VXORPS __m512 _mm512_mask_xor_ps (__m512 a, __mmask16 m, __m512 b);
VXORPS __m512 _mm512_maskz_xor_ps (__mmask16 m, __m512 a);
VXORPS __m256 _mm256_xor_ps (__m256 a, __m256 b);
VXORPS __m256 _mm256_mask_xor_ps (__m256 a, __mmask8 m, __m256 b);
VXORPS __m256 _mm256_maskz_xor_ps (__mmask8 m, __m256 a);
XORPS __m128 _mm_xor_ps (__m128 a, __m128 b);
VXORPS __m128 _mm_mask_xor_ps (__m128 a, __mmask8 m, __m128 b);
VXORPS __m128 _mm_maskz_xor_ps (__mmask8 m, __m128 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instructions, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-49, "Type E4 Class Exception Conditions."

XRESLDTRK—Resume Tracking Load Addresses

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 01 E9 XRESLDTRK	ZO	V/V	TSXLDTRK	Specifies the end of an Intel TSX suspend read address tracking region.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A	N/A

Description

The instruction marks the end of an Intel TSX (RTM) suspend load address tracking region. If the instruction is used inside a suspend load address tracking region it will end the suspend region and all following load addresses will be added to the transaction read set. If this instruction is used inside an active transaction but not in a suspend region it will cause transaction abort.

If the instruction is used outside of a transactional region it behaves like a NOP.

Chapter 16, "Programming with Intel® Transactional Synchronization Extensions," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 provides additional information on Intel® TSX Suspend Load Address Tracking.

Operation

XRESLDTRK

```
IF RTM_ACTIVE = 1:
    IF SUSLDTRK_ACTIVE = 1:
        SUSLDTRK_ACTIVE := 0
    ELSE:
        RTM_ABORT
ELSE:
    NOP
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XRESLDTRK void _xresldtrk(void);

SIMD Floating-Point Exceptions

None.

Other Exceptions

#UD	If CPUID.07H.00H:EDX.TSXLDTRK[16] = 0.
	If the LOCK prefix is used.

XRSTOR—Restore Processor Extended States

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F AE /5 XRSTOR mem	M	V/V	XSAVE	Restore state components specified by EDX:EAX from mem.
NP REX.W + 0F AE /5 XRSTOR64 mem	M	V/N.E.	XSAVE	Restore state components specified by EDX:EAX from mem.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

Description

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1. Like FXRSTOR and FXSAVE, the memory format used for x87 state depends on a REX.W prefix; see Section 13.5.1, "x87 State" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Section 13.8, "Operation of XRSTOR," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTOR may take one of two forms: standard and compacted. Bit 63 of the XCOMP_BV field in the XSAVE header determines which form is used: value 0 specifies the standard form, while value 1 specifies the compacted form.
- If RFBM[*i*] = 0, XRSTOR does not update state component *i*.[1]
- If RFBM[*i*] = 1 and bit *i* is clear in the XSTATE_BV field in the XSAVE header, XRSTOR initializes state component *i*.
- If RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1, XRSTOR loads state component *i* from the XSAVE area.
- The standard form of XRSTOR treats MXCSR (which is part of state component 1 — SSE) differently from the XMM registers. If either form attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTOR loads the internal value XRSTOR_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTOR, the processor tracks as in-use (not in initial configuration) any state component *i* for which RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1; it tracks as modified any state component *i* for which RFBM[*i*] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, "Processor Tracking of XSAVE-Managed State," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 for discussion of the bitmaps XINUSE and XMODIFIED and of the quantity XRSTOR_INFO.

1. There is an exception if RFBM[1] = 0 and RFBM[2] = 1. In this case, the standard form of XRSTOR will load MXCSR from memory, even though MXCSR is part of state component 1 — SSE. The compacted form of XRSTOR does not make this exception.

Operation

```
RFBM := XCR0 AND EDX:EAX;    /* bitwise logical AND */
COMPMASK := XCOMP_BV field from XSAVE header;
RSTORMASK := XSTATE_BV field from XSAVE header;

IF COMPMASK[63] = 0
    THEN
        /* Standard form of XRSTOR */
        TO_BE_RESTORED := RFBM AND RSTORMASK;
        TO_BE_INITIALIZED := RFBM AND NOT RSTORMASK;

        IF TO_BE_RESTORED[0] = 1
            THEN
                XINUSE[0] := 1;
                load x87 state from legacy region of XSAVE area;
        ELSIF TO_BE_INITIALIZED[0] = 1
            THEN
                XINUSE[0] := 0;
                initialize x87 state;
        FI;

        IF RFBM[1] = 1 OR RFBM[2] = 1
            THEN load MXCSR from legacy region of XSAVE area;
        FI;

        IF TO_BE_RESTORED[1] = 1
            THEN
                XINUSE[1] := 1;
                load XMM registers from legacy region of XSAVE area; // this step does not load MXCSR
        ELSIF TO_BE_INITIALIZED[1] = 1
            THEN
                XINUSE[1] := 0;
                set all XMM registers to 0; // this step does not initialize MXCSR
        FI;

        FOR i := 2 TO 62
            IF TO_BE_RESTORED[i] = 1
                THEN
                    XINUSE[i] := 1;
                    load XSAVE state component i at offset n from base of XSAVE area;
                        // n enumerated by CPUID.0DH.i:EBX)
            ELSIF TO_BE_INITIALIZED[i] = 1
                THEN
                    XINUSE[i] := 0;
                    initialize XSAVE state component i;
            FI;
        ENDFOR;

    ELSE
        /* Compacted form of XRSTOR */
        IF CPUID.0DH.01H:EAX.XSAVEC[1] = 0
            THEN    /* compacted form not supported */
                #GP(0);
        FI;
```

```
        FORMAT = COMPMASK AND 7FFFFFFF_FFFFFFFFH;
        RESTORE_FEATURES = FORMAT AND RFBM;
        TO_BE_RESTORED := RESTORE_FEATURES AND RSTORMASK;
        FORCE_INIT := RFBM AND NOT FORMAT;
        TO_BE_INITIALIZED = (RFBM AND NOT RSTORMASK) OR FORCE_INIT;

        IF TO_BE_RESTORED[0] = 1
            THEN
                XINUSE[0] := 1;
                load x87 state from legacy region of XSAVE area;
        ELSIF TO_BE_INITIALIZED[0] = 1
            THEN
                XINUSE[0] := 0;
                initialize x87 state;
        FI;

        IF TO_BE_RESTORED[1] = 1
            THEN
                XINUSE[1] := 1;
                load SSE state from legacy region of XSAVE area; // this step loads the XMM registers and MXCSR
        ELSIF TO_BE_INITIALIZED[1] = 1
            THEN
                set all XMM registers to 0;
                XINUSE[1] := 0;
                MXCSR := 1F80H;
        FI;

        NEXT_FEATURE_OFFSET = 576;          // Legacy area and XSAVE header consume 576 bytes
        FOR i := 2 TO 62
            IF FORMAT[i] = 1
                THEN
                    IF TO_BE_RESTORED[i] = 1
                        THEN
                            XINUSE[i] := 1;
                            load XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
                    FI;
NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID.0DH.i:EAX);
            FI;
            IF TO_BE_INITIALIZED[i] = 1
                THEN
                    XINUSE[i] := 0;
                    initialize XSAVE state component i;
            FI;
        ENDFOR;
FI;

XMODIFIED := NOT RFBM;

IF in VMX non-root operation
    THEN VMXNR := 1;
    ELSE VMXNR := 0;
FI;
LAXA := linear address of XSAVE area;
```

XRSTOR_INFO := ⟨CPL,VMXNR,LAXA,COMPMASK⟩;

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XRSTOR void _xrstor(void * , unsigned __int64);
XRSTOR void _xrstor64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.0DH.01H:EAX.XSAVEC[1] = 0.
	If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.
	If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.
	If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.0DH.01H:EAX.XSAVEC[1] = 0.
	If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.
	If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.
	If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.

	If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If a memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.

If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.0DH.01H:EAX.XSAVEC[1] = 0.

	If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.
	If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.
	If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XRSTORS—Restore Processor Extended States Supervisor

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C7 /3 XRSTORS mem	M	V/V	XSS	Restore state components specified by EDX:EAX from mem.
NP REX.W + 0F C7 /3 XRSTORS64 mem	M	V/N.E.	XSS	Restore state components specified by EDX:EAX from mem.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

Description

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32_XSS MSR. XRSTORS may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1. Like FXRSTOR and FXSAVE, the memory format used for x87 state depends on a REX.W prefix; see Section 13.5.1, "x87 State" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Section 13.12, "Operation of XRSTORS," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTORS is similar to that of the compacted form of XRSTOR; XRSTORS cannot restore from an XSAVE area in which the extended region is in the standard format (see Section 13.4.3, "Extended Region of an XSAVE Area" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1).
- XRSTORS differs from XRSTOR in that it can restore state components corresponding to bits set in the IA32_XSS MSR.
- If RFBM[*i*] = 0, XRSTORS does not update state component *i*.
- If RFBM[*i*] = 1 and bit *i* is clear in the XSTATE_BV field in the XSAVE header, XRSTORS initializes state component *i*.
- If RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1, XRSTORS loads state component *i* from the XSAVE area.
- If XRSTORS attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTORS loads the internal value XRSTOR_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTORS, the processor tracks as in-use (not in initial configuration) any state component *i* for which RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1; it tracks as modified any state component *i* for which RFBM[*i*] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, "Processor Tracking of XSAVE-Managed State," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 for discussion of the bitmaps XINUSE and XMODIFIED and of the quantity XRSTOR_INFO.

Operation

```
RFBM := (XCR0 OR IA32_XSS) AND EDX:EAX;              /* bitwise logical OR and AND */
COMPMASK := XCOMP_BV field from XSAVE header;
RSTORMASK := XSTATE_BV field from XSAVE header;

FORMAT = COMPMASK AND 7FFFFFFF_FFFFFFFFH;
RESTORE_FEATURES = FORMAT AND RFBM;
TO_BE_RESTORED := RESTORE_FEATURES AND RSTORMASK;
FORCE_INIT := RFBM AND NOT FORMAT;
TO_BE_INITIALIZED = (RFBM AND NOT RSTORMASK) OR FORCE_INIT;

IF TO_BE_RESTORED[0] = 1
    THEN
        XINUSE[0] := 1;
        load x87 state from legacy region of XSAVE area;
ELSIF TO_BE_INITIALIZED[0] = 1
    THEN
        XINUSE[0] := 0;
        initialize x87 state;
FI;

IF TO_BE_RESTORED[1] = 1
    THEN
        XINUSE[1] := 1;
        load SSE state from legacy region of XSAVE area; // this step loads the XMM registers and MXCSR
ELSIF TO_BE_INITIALIZED[1] = 1
    THEN
        set all XMM registers to 0;
        XINUSE[1] := 0;
        MXCSR := 1F80H;
FI;

NEXT_FEATURE_OFFSET = 576;          // Legacy area and XSAVE header consume 576 bytes
FOR i := 2 TO 62
    IF FORMAT[i] = 1
        THEN
            IF TO_BE_RESTORED[i] = 1
                THEN
                    XINUSE[i] := 1;
                    load XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
            FI;
NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID.0DH.i:EAX);
    FI;
    IF TO_BE_INITIALIZED[i] = 1
        THEN
            XINUSE[i] := 0;
            initialize XSAVE state component i;
    FI;
ENDFOR;

XMODIFIED := NOT RFBM;

IF in VMX non-root operation
    THEN VMXNR := 1;
```

```
    ELSE VMXNR := 0;
FI;
LAXA := linear address of XSAVE area;
XRSTOR_INFO := ⟨CPL,VMXNR,LAXA,COMPMASK⟩;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XRSTORS void _xrstors(void * , unsigned __int64);
XRSTORS64 void _xrstors64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If CPL > 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
	If a bit in XCR0|IA32_XSS is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[26] = 0 or CPUID.0DH.01H:EAX.XSAVES[3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
	If a bit in XCR0|IA32_XSS is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[26] = 0 or CPUID.0DH.01H:EAX.XSAVES[3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If CPL > 0.
	If a memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
	If a bit in XCR0|IA32_XSS is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[26] = 0 or CPUID.0DH.01H:EAX.XSAVES[3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

XSAVE—Save Processor Extended States

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F AE /4 XSAVE mem	M	V/V	XSAVE	Save state components specified by EDX:EAX to mem.
NP REX.W + 0F AE /4 XSAVE64 mem	M	V/N.E.	XSAVE	Save state components specified by EDX:EAX to mem.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	N/A	N/A	N/A

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1. Like FXRSTOR and FXSAVE, the memory format used for x87 state depends on a REX.W prefix; see Section 13.5.1, "x87 State" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Section 13.7, "Operation of XSAVE," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 provides a detailed description of the operation of the XSAVE instruction. The following items provide a high-level outline:

- XSAVE saves state component *i* if and only if RFBM[*i*] = 1.[1]
- XSAVE does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1).
- XSAVE reads the XSTATE_BV field of the XSAVE header (see Section 13.4.2, "XSAVE Header" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1) and writes a modified value back to memory as follows. If RFBM[*i*] = 1, XSAVE writes XSTATE_BV[*i*] with the value of XINUSE[*i*]. (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.) If RFBM[*i*] = 0, XSAVE writes XSTATE_BV[*i*] with the value that it read from memory (it does not modify the bit). XSAVE does not write to any part of the XSAVE header other than the XSTATE_BV field.
- XSAVE always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

1. An exception is made for MXCSR and MXCSR_MASK, which belong to state component 1 — SSE. XSAVE saves these values to memory if either RFBM[1] or RFBM[2] is 1.

Operation

```
RFBM := XCR0 AND EDX:EAX;    /* bitwise logical AND */
OLD_BV := XSTATE_BV field from XSAVE header;

IF RFBM[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF RFBM[1] = 1
    THEN store XMM registers into legacy region of XSAVE area; // this step does not save MXCSR or MXCSR_MASK
FI;

IF RFBM[1] = 1 OR RFBM[2] = 1
    THEN store MXCSR and MXCSR_MASK into legacy region of XSAVE area;
FI;

FOR i := 2 TO 62
    IF RFBM[i] = 1
        THEN save XSAVE state component i at offset n from base of XSAVE area (n enumerated by CPUID.0DH.i:EBX);
    FI;
ENDFOR;

XSTATE_BV field in XSAVE header := (OLD_BV AND NOT RFBM) OR (XINUSE AND RFBM);
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSAVE void _xsave(void * , unsigned __int64);
XSAVE void _xsave64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XSAVEC—Save Processor Extended States With Compaction

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C7 /4 XSAVEC mem	M	V/V	XSAVEC	Save state components specified by EDX:EAX to mem with compaction.
NP REX.W + 0F C7 /4 XSAVEC64 mem	M	V/N.E.	XSAVEC	Save state components specified by EDX:EAX to mem with compaction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	N/A	N/A	N/A

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1. Like FXRSTOR and FXSAVE, the memory format used for x87 state depends on a REX.W prefix; see Section 13.5.1, "x87 State" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Section 13.10, "Operation of XSAVEC," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 provides a detailed description of the operation of the XSAVEC instruction. The following items provide a high-level outline:

- Execution of XSAVEC is similar to that of XSAVE. XSAVEC differs from XSAVE in that it uses compaction and that it may use the init optimization.
- XSAVEC saves state component *i* if and only if RFBM[*i*] = 1 and XINUSE[*i*] = 1.[1] (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.)
- XSAVEC does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1).
- XSAVEC writes the logical AND of RFBM and XINUSE to the XSTATE_BV field of the XSAVE header.[2,3] (See Section 13.4.2, "XSAVE Header" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.) XSAVEC sets bit 63 of the XCOMP_BV field and sets bits 62:0 of that field to RFBM[62:0]. XSAVEC does not write to any parts of the XSAVE header other than the XSTATE_BV and XCOMP_BV fields.
- XSAVEC always uses the compacted format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

1. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVEC saves SSE state as long as RFBM[1] = 1.
2. Unlike XSAVE and XSAVEOPT, XSAVEC clears bits in the XSTATE_BV field that correspond to bits that are clear in RFBM.
3. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVEC sets XSTATE_BV[1] to 1 as long as RFBM[1] = 1.

Operation

```
RFBM := XCR0 AND EDX:EAX;              /* bitwise logical AND */
TO_BE_SAVED := RFBM AND XINUSE;        /* bitwise logical AND */
If MXCSR ≠ 1F80H AND RFBM[1]
    TO_BE_SAVED[1] = 1;
FI;

IF TO_BE_SAVED[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF TO_BE_SAVED[1] = 1
    THEN store SSE state into legacy region of XSAVE area; // this step saves the XMM registers, MXCSR, and MXCSR_MASK
FI;

NEXT_FEATURE_OFFSET = 576;             // Legacy area and XSAVE header consume 576 bytes
FOR i := 2 TO 62
    IF RFBM[i] = 1
        THEN
            IF TO_BE_SAVED[i]
                THEN save XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
            FI;
            NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID.0DH.i:EAX);
    FI;
ENDFOR;

XSTATE_BV field in XSAVE header := TO_BE_SAVED;
XCOMP_BV field in XSAVE header := RFBM OR 80000000_00000000H;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSAVEC void _xsavec(void * , unsigned __int64);
XSAVEC64 void _xsavec64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[26] = 0 or CPUID.0DH.01H:EAX.XSAVEC[1] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[26] = 0 or CPUID.0DH.01H:EAX.XSAVEC[1] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[26] = 0 or CPUID.0DH.01H:EAX.XSAVEC[1] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XSAVEOPT—Save Processor Extended States Optimized

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F AE /6 XSAVEOPT mem	M	V/V	XSAVEOPT	Save state components specified by EDX:EAX to mem, optimizing if possible.
NP REX.W + 0F AE /6 XSAVEOPT64 mem	M	V/V	XSAVEOPT	Save state components specified by EDX:EAX to mem, optimizing if possible.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	N/A	N/A	N/A

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1. Like FXRSTOR and FXSAVE, the memory format used for x87 state depends on a REX.W prefix; see Section 13.5.1, "x87 State" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Section 13.9, "Operation of XSAVEOPT," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 provides a detailed description of the operation of the XSAVEOPT instruction. The following items provide a high-level outline:

- Execution of XSAVEOPT is similar to that of XSAVE. XSAVEOPT differs from XSAVE in that it may use the init and modified optimizations. The performance of XSAVEOPT will be equal to or better than that of XSAVE.
- XSAVEOPT saves state component *i* only if RFBM[*i*] = 1 and XINUSE[*i*] = 1.[1] (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.) Even if both bits are 1, XSAVEOPT may optimize and not save state component *i* if (1) state component *i* has not been modified since the last execution of XRSTOR or XRSTORS; and (2) this execution of XSAVES corresponds to that last execution of XRSTOR or XRSTORS as determined by the internal value XRSTOR_INFO (see the Operation section below).
- XSAVEOPT does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1).
- XSAVEOPT reads the XSTATE_BV field of the XSAVE header (see Section 13.4.2, "XSAVE Header," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1) and writes a modified value back to memory as follows. If RFBM[*i*] = 1, XSAVEOPT writes XSTATE_BV[*i*] with the value of XINUSE[*i*]. If RFBM[*i*] = 0, XSAVEOPT writes XSTATE_BV[*i*] with the value that it read from memory (it does not modify the bit). XSAVEOPT does not write to any part of the XSAVE header other than the XSTATE_BV field.
- XSAVEOPT always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) will result in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

1. There is an exception made for MXCSR and MXCSR_MASK, which belong to state component 1 — SSE. XSAVEOPT always saves these to memory if RFBM[1] = 1 or RFBM[2] = 1, regardless of the value of XINUSE.

See Section 13.6, "Processor Tracking of XSAVE-Managed State," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 for discussion of the bitmap XMODIFIED and of the quantity XRSTOR_INFO.

Operation

```
RFBM := XCR0 AND EDX:EAX;    /* bitwise logical AND */
OLD_BV := XSTATE_BV field from XSAVE header;
TO_BE_SAVED := RFBM AND XINUSE;

IF in VMX non-root operation
    THEN VMXNR := 1;
    ELSE VMXNR := 0;
FI;
LAXA := linear address of XSAVE area;
IF XRSTOR_INFO = ⟨CPL,VMXNR,LAXA,00000000_00000000H⟩
    THEN TO_BE_SAVED := TO_BE_SAVED AND XMODIFIED;
FI;

IF TO_BE_SAVED[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF TO_BE_SAVED[1]
    THEN store XMM registers into legacy region of XSAVE area; // this step does not save MXCSR or MXCSR_MASK
FI;

IF RFBM[1] = 1 or RFBM[2] = 1
    THEN store MXCSR and MXCSR_MASK into legacy region of XSAVE area;
FI;

FOR i := 2 TO 62
    IF TO_BE_SAVED[i] = 1
        THEN save XSAVE state component i at offset n from base of XSAVE area (n enumerated by CPUID.0DH.i:EBX);
    FI;
ENDFOR;

XSTATE_BV field in XSAVE header := (OLD_BV AND NOT RFBM) OR (XINUSE AND RFBM);
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

```
XSAVEOPT void _xsaveopt( void * , unsigned __int64);
XSAVEOPT void _xsaveopt64( void * , unsigned __int64);
```

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[26] = 0 or CPUID.0DH.01H:EAX.XSAVEOPT[0] = 0.

	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[26] = 0 or CPUID.0DH.01H:EAX.XSAVEOPT[0] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[26] = 0 or CPUID.0DH.01H:EAX.XSAVEOPT[0] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XSAVES—Save Processor Extended States Supervisor

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C7 /5 XSAVES mem	M	V/V	XSS	Save state components specified by EDX:EAX to mem with compaction, optimizing if possible.
NP REX.W + 0F C7 /5 XSAVES64 mem	M	V/N.E.	XSS	Save state components specified by EDX:EAX to mem with compaction, optimizing if possible.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	N/A	N/A	N/A

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32_XSS MSR. XSAVES may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1. Like FXRSTOR and FXSAVE, the memory format used for x87 state depends on a REX.W prefix; see Section 13.5.1, "x87 State," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Section 13.11, "Operation of XSAVES," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 provides a detailed description of the operation of the XSAVES instruction. The following items provide a high-level outline:

- Execution of XSAVES is similar to that of XSAVEC. XSAVES differs from XSAVEC in that it can save state components corresponding to bits set in the IA32_XSS MSR and that it may use the modified optimization.
- XSAVES saves state component *i* only if RFBM[*i*] = 1 and XINUSE[*i*] = 1.[1] (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.) Even if both bits are 1, XSAVES may optimize and not save state component *i* if (1) state component *i* has not been modified since the last execution of XRSTOR or XRSTORS; and (2) this execution of XSAVES correspond to that last execution of XRSTOR or XRSTORS as determined by XRSTOR_INFO (see the Operation section below).
- XSAVES does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1).
- XSAVES writes the logical AND of RFBM and XINUSE to the XSTATE_BV field of the XSAVE header.[2] (See Section 13.4.2, "XSAVE Header," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.) XSAVES sets bit 63 of the XCOMP_BV field and sets bits 62:0 of that field to RFBM[62:0]. XSAVES does not write to any parts of the XSAVE header other than the XSTATE_BV and XCOMP_BV fields.
- XSAVES always uses the compacted format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

---

1. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, the init optimization does not apply and XSAVEC will save SSE state as long as RFBM[1] = 1 and the modified optimization is not being applied.
2. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVES sets XSTATE_BV[1] to 1 as long as RFBM[1] = 1.

See Section 13.6, "Processor Tracking of XSAVE-Managed State," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 for discussion of the bitmap XMODIFIED and of the quantity XRSTOR_INFO.

Operation

```
RFBM := (XCR0 OR IA32_XSS) AND EDX:EAX;              /* bitwise logical OR and AND */
IF in VMX non-root operation
    THEN VMXNR := 1;
    ELSE VMXNR := 0;
FI;
LAXA := linear address of XSAVE area;
COMPMASK := RFBM OR 80000000_00000000H;
TO_BE_SAVED := RFBM AND XINUSE;
IF XRSTOR_INFO = ⟨CPL,VMXNR,LAXA,COMPMASK⟩
    THEN TO_BE_SAVED := TO_BE_SAVED AND XMODIFIED;
FI;
IF MXCSR ≠ 1F80H AND RFBM[1]
    THEN TO_BE_SAVED[1] = 1;
FI;

IF TO_BE_SAVED[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF TO_BE_SAVED[1] = 1
    THEN store SSE state into legacy region of XSAVE area; // this step saves the XMM registers, MXCSR, and MXCSR_MASK
FI;

NEXT_FEATURE_OFFSET = 576;          // Legacy area and XSAVE header consume 576 bytes
FOR i := 2 TO 62
    IF RFBM[i] = 1
        THEN
            IF TO_BE_SAVED[i]
                THEN
                    save XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
                    IF i = 8        // state component 8 is for PT state
                        THEN IA32_RTIT_CTL.TraceEn[bit 0] := 0;
                    FI;
            FI;
            NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID.0DH.i:EAX);
    FI;
ENDFOR;

NEW_HEADER := RFBM AND XINUSE;
IF MXCSR ≠ 1F80H AND RFBM[1]
    THEN NEW_HEADER[1] = 1;
FI;
XSTATE_BV field in XSAVE header := NEW_HEADER;
XCOMP_BV field in XSAVE header := COMPMASK;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSAVES void _xsaves(void * , unsigned __int64);
XSAVES64 void _xsaves64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If CPL > 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[26] = 0 or CPUID.0DH.01H:EAX.XSAVES[3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[26] = 0 or CPUID.0DH.01H:EAX.XSAVES[3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If CPL > 0.
	If the memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[26] = 0 or CPUID.0DH.01H:EAX.XSAVES[3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

XSETBV—Set Extended Control Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F 01 D1	XSETBV	ZO	Valid	Valid	Write the value in EDX:EAX to the XCR specified by ECX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Writes the contents of registers EDX:EAX into the 64-bit extended control register (XCR) specified in the ECX register. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The contents of the EDX register are copied to high-order 32 bits of the selected XCR and the contents of the EAX register are copied to low-order 32 bits of the XCR. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are ignored.) Undefined or reserved bits in an XCR should be set to values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. Specifying a reserved or unimplemented XCR in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write to reserved bits in an XCR.

Currently, only XCR0 is supported. Thus, all other values of ECX are reserved and will cause a #GP(0). Note that bit 0 of XCR0 (corresponding to x87 state) must be set to 1; the instruction will cause a #GP(0) if an attempt is made to clear this bit. In addition, the instruction causes a #GP(0) if an attempt is made to set XCR0[2] (AVX state) while clearing XCR0[1] (SSE state); it is necessary to set both bits to use AVX instructions; Section 13.3, "Enabling the XSAVE Feature Set and XSAVE-Enabled Features," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Operation

```
XCR[ECX] := EDX:EAX;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSETBV void _xsetbv(unsigned int, unsigned __int64);

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If an invalid XCR is specified in ECX.
	If the value in EDX:EAX sets bits that are reserved in the XCR specified by ECX.
	If an attempt is made to clear bit 0 of XCR0.
	If an attempt is made to set XCR0[2:1] to 10b.
#UD	If CPUID.01H:ECX.XSAVE[26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If an invalid XCR is specified in ECX.
	If the value in EDX:EAX sets bits that are reserved in the XCR specified by ECX.
	If an attempt is made to clear bit 0 of XCR0.
	If an attempt is made to set XCR0[2:1] to 10b.
#UD	If CPUID.01H:ECX.XSAVE[26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	The XSETBV instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

XSUSLDTRK—Suspend Tracking Load Addresses

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 01 E8 XSUSLDTRK	ZO	V/V	TSXLDTRK	Specifies the start of an Intel TSX suspend read address tracking region.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A	N/A

Description

The instruction marks the start of an Intel TSX (RTM) suspend load address tracking region. If the instruction is used inside a transactional region, subsequent loads are not added to the read set of the transaction. If the instruction is used inside a suspend load address tracking region it will cause transaction abort.

If the instruction is used outside of a transactional region it behaves like a NOP.

Chapter 16, "Programming with Intel® Transactional Synchronization Extensions," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 provides additional information on Intel® TSX Suspend Load Address Tracking.

Operation

XSUSLDTRK

```
IF RTM_ACTIVE = 1:
    IF SUSLDTRK_ACTIVE = 0:
        SUSLDTRK_ACTIVE := 1
    ELSE:
        RTM_ABORT
ELSE:
    NOP
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSUSLDTRK void _xsusldtrk(void);

SIMD Floating-Point Exceptions

None.

Other Exceptions

#UD	If CPUID.07H.00H:EDX.TSXLDTRK[16] = 0.
	If the LOCK prefix is used.

XTEST—Test if in Transactional Execution

Opcode/Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 D6 XTEST	ZO	V/V	HLE or RTM	Test if executing in a transactional region.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
ZO	N/A	N/A	N/A	N/A

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	If CPUID.07H.00H:EBX.HLE[4] = 0 and CPUID.07H.00H:EBX.RTM[11] = 0. If LOCK prefix is used.

CHAPTER 7
SAFER MODE EXTENSIONS REFERENCE

7.1 OVERVIEW

This chapter describes the Safer Mode Extensions (SMX) for the Intel 64 and IA-32 architectures. Safer Mode Extensions (SMX) provide a programming interface for system software to establish a measured environment within the platform to support trust decisions by end users. The measured environment includes:

- Measured launch of a system executive, referred to as a Measured Launched Environment (MLE)[1]. The system executive may be based on a Virtual Machine Monitor (VMM), a measured VMM is referred to as MVMM[2].
- Mechanisms to ensure the above measurement is protected and stored in a secure location in the platform.
- Protection mechanisms that allow the VMM to control attempts to modify the VMM.

The measurement and protection mechanisms used by a measured environment are supported by the capabilities of an Intel® Trusted Execution Technology (Intel® TXT) platform:

- The SMX are the processor's programming interface in an Intel TXT platform.
- The chipset in an Intel TXT platform provides enforcement of the protection mechanisms.
- Trusted Platform Module (TPM) 1.2 in the platform provides platform configuration registers (PCRs) to store software measurement values.

7.2 SMX FUNCTIONALITY

SMX functionality is provided in an Intel 64 processor through the GETSEC instruction via leaf functions. The GETSEC instruction supports multiple leaf functions. Leaf functions are selected by the value in EAX at the time GETSEC is executed. Each GETSEC leaf function is documented separately in the reference pages with a unique mnemonic (even though these mnemonics share the same opcode, 0F 37).

7.2.1 Detecting and Enabling SMX

Software can detect support for SMX operation using the CPUID instruction. If software executes CPUID with 1 in EAX, a value of 1 in bit 6 of ECX indicates support for SMX operation (GETSEC is available), see CPUID instruction for the layout of feature flags of reported by CPUID.01H:ECX.

System software enables SMX operation by setting CR4.SMXE[Bit 14] = 1 before attempting to execute GETSEC. Otherwise, execution of GETSEC results in the processor signaling an invalid opcode exception (#UD).

If the CPUID SMX feature flag is clear (CPUID.01H:ECX[6] = 0), attempting to set CR4.SMXE[Bit 14] results in a general protection exception.

The IA32_FEATURE_CONTROL MSR (at address 03AH) provides feature control bits that configure operation of VMX and SMX. These bits are documented in Table 7-1.

Table 7-1. Layout of IA32_FEATURE_CONTROL

Bit Position	Description
0	Lock bit (0 = unlocked, 1 = locked). When set to '1' further writes to this MSR are blocked.
1	Enable VMX in SMX operation.
2	Enable VMX outside SMX operation.

1. See the Intel® Trusted Execution Technology Measured Launched Environment Programming Guide.
2. An MVMM is sometimes referred to as a measured launched environment (MLE). See the Intel® Trusted Execution Technology Measured Launched Environment Programming Guide.

Table 7-1. Layout of IA32_FEATURE_CONTROL

7:3	Reserved
14:8	SENTER Local Function Enables: When set, each bit in the field represents an enable control for a corresponding SENTER function.
15	SENTER Global Enable: Must be set to '1' to enable operation of GETSEC[SENTER].
16	Reserved
17	SGX Launch Control Enable: Must be set to '1' to enable runtime re-configuration of SGX Launch Control via the IA32_SGXLEPUBKEYHASHn MSR.
18	SGX Global Enable: Must be set to '1' to enable Intel SGX leaf functions.
19	Reserved
20	LMCE On: When set, system software can program the MSRs associated with LMCE to configure delivery of some machine check exceptions to a single logical processor.
63:21	Reserved

- Bit 0 is a lock bit. If the lock bit is clear, an attempt to execute VMXON will cause a general-protection exception. Attempting to execute GETSEC[SENTER] when the lock bit is clear will also cause a general-protection exception. If the lock bit is set, WRMSR to the IA32_FEATURE_CONTROL MSR will cause a general-protection exception. Once the lock bit is set, the MSR cannot be modified until a power-on reset. System BIOS can use this bit to provide a setup option for BIOS to disable support for VMX, SMX or both VMX and SMX.
- Bit 1 enables VMX in SMX operation (between executing the SENTER and SEXIT leaves of GETSEC). If this bit is clear, an attempt to execute VMXON in SMX will cause a general-protection exception if executed in SMX operation. Attempts to set this bit on logical processors that do not support both VMX operation (Chapter 7, "Safer Mode Extensions Reference") and SMX operation cause general-protection exceptions.
- Bit 2 enables VMX outside SMX operation. If this bit is clear, an attempt to execute VMXON will cause a general-protection exception if executed outside SMX operation. Attempts to set this bit on logical processors that do not support VMX operation cause general-protection exceptions.
- Bits 8 through 14 specify enabled functionality of the SENTER leaf function. Each bit in the field represents an enable control for a corresponding SENTER function. Only enabled SENTER leaf functionality can be used when executing SENTER.
- Bits 15 specify global enable of all SENTER functionalities.

7.2.2 SMX Instruction Summary

System software must first query for available GETSEC leaf functions by executing GETSEC[CAPABILITIES]. The CAPABILITIES leaf function returns a bit map of available GETSEC leaves. An attempt to execute an unsupported leaf index results in an undefined opcode (#UD) exception.

7.2.2.1 GETSEC[CAPABILITIES]

The SMX functionality provides an architectural interface for newer processor generations to extend SMX capabilities. Specifically, the GETSEC instruction provides a capability leaf function for system software to discover the available GETSEC leaf functions that are supported in a processor. Table 7-2 lists the currently available GETSEC leaf functions.

Table 7-2. GETSEC Leaf Functions

Index (EAX)	Leaf function	Description
0	CAPABILITIES	Returns the available leaf functions of the GETSEC instruction.
1	Undefined	Reserved
2	ENTERACCS	Enter
3	EXITAC	Exit
4	SENTER	Launch an MLE.
5	SEXIT	Exit the MLE.
6	PARAMETERS	Return SMX related parameter information.
7	SMCTRL	SMX mode control.
8	WAKEUP	Wake up sleeping processors in safer mode.
9 - (4G-1)	Undefined	Reserved

7.2.2.2 GETSEC[ENTERACCS]

The GETSEC[ENTERACCS] leaf enables authenticated code execution mode. The ENTERACCS leaf function performs an authenticated code module load using the chipset public key as the signature verification. ENTERACCS requires the existence of an Intel® Trusted Execution Technology capable chipset since it unlocks the chipset private configuration register space after successful authentication of the loaded module. The physical base address and size of the authenticated code module are specified as input register values in EBX and ECX, respectively.

While in the authenticated code execution mode, certain processor state properties change. For this reason, the time in which the processor operates in authenticated code execution mode should be limited to minimize impact on external system events.

Upon entry into, the previous paging context is disabled (since the authenticated code module image is specified with physical addresses and can no longer rely upon external memory-based page-table structures).

Prior to executing the GETSEC[ENTERACCS] leaf, system software must ensure the logical processor issuing GETSEC[ENTERACCS] is the boot-strap processor (BSP), as indicated by IA32_APIC_BASE.BSP = 1. System software must ensure other logical processors are in a suitable idle state and not marked as BSP.

The GETSEC[ENTERACCS] leaf may be used by different agents to load different authenticated code modules to perform functions related to different aspects of a measured environment, for example system software and Intel® TXT enabled BIOS may use more than one authenticated code modules.

7.2.2.3 GETSEC[EXITAC]

GETSEC[EXITAC] takes the processor out of authenticated code execution mode. When this instruction leaf is executed, the contents of the authenticated code execution area are scrubbed and control is transferred to the non-authenticated context defined by a near pointer passed with the GETSEC[EXITAC] instruction.

The authenticated code execution area is no longer accessible after completion of GETSEC[EXITAC]. RBX (or EBX) holds the address of the near absolute indirect target to be taken.

7.2.2.4 GETSEC[SENTER]

The GETSEC[SENTER] leaf function is used by the initiating logical processor (ILP) to launch an MLE. GETSEC[SENTER] can be considered a superset of the ENTERACCS leaf, because it enters as part of the measured environment launch.

Measured environment startup consists of the following steps:

- the ILP rendezvous the responding logical processors (RLPs) in the platform into a controlled state (At the completion of this handshake, all the RLPs except for the ILP initiating the measured environment launch are placed in a newly defined SENTER sleep state).
- Load and authenticate the authenticated code module required by the measured environment, and enter authenticated code execution mode.
- Verify and lock certain system configuration parameters.
- Measure the dynamic root of trust and store into the PCRs in TPM.
- Transfer control to the MLE with interrupts disabled.

Prior to executing the GETSEC[SENTER] leaf, system software must ensure the platform's TPM is ready for access and the ILP is the boot-strap processor (BSP), as indicated by IA32_APIC_BASE.BSP. System software must ensure other logical processors (RLPs) are in a suitable idle state and not marked as BSP.

System software launching a measurement environment is responsible for providing a proper authenticate code module address when executing GETSEC[SENTER]. The AC module responsible for the launch of a measured environment and loaded by GETSEC[SENTER] is referred to as SINIT. See *Intel® Trusted Execution Technology Measured Launched Environment Programming Guide* for additional information on system software requirements prior to executing GETSEC[SENTER].

7.2.2.5 GETSEC[SEXIT]

System software exits the measured environment by executing the instruction GETSEC[SEXIT] on the ILP. This instruction rendezvous the responding logical processors in the platform for exiting from the measured environment. External events (if left masked) are unmasked and Intel® TXT-capable chipset's private configuration space is re-locked.

7.2.2.6 GETSEC[PARAMETERS]

The GETSEC[PARAMETERS] leaf function is used to report attributes, options, and limitations of SMX operation. Software uses this leaf to identify operating limits or additional options.

The information reported by GETSEC[PARAMETERS] may require executing the leaf multiple times using EBX as an index. If the GETSEC[PARAMETERS] instruction leaf or if a specific parameter field is not available, then SMX operation should be interpreted to use the default limits of respective GETSEC leaves or parameter fields defined in the GETSEC[PARAMETERS] leaf.

7.2.2.7 GETSEC[SMCTRL]

The GETSEC[SMCTRL] leaf function is used for providing additional control over specific conditions associated with the SMX architecture. An input register is supported for selecting the control operation to be performed. See the specific leaf description for details on the type of control provided.

7.2.2.8 GETSEC[WAKEUP]

Responding logical processors (RLPs) are placed in the SENTER sleep state after the initiating logical processor executes GETSEC[SENTER]. The ILP can wake up RLPs to join the measured environment by using GETSEC[WAKEUP]. When the RLPs in SENTER sleep state wake up, these logical processors begin execution at the entry point defined in a data structure held in system memory (pointed to by an chipset register LT.MLE.JOIN) in TXT configuration space.

7.2.3 Measured Environment and SMX

This section gives a simplified view of a representative life cycle of a measured environment that is launched by a system executive using SMX leaf functions. The *Intel® Trusted Execution Technology Measured Launched Environment Programming Guide* provides more detailed examples of using SMX and chipset resources (including chipset registers, Trusted Platform Module) to launch an MVMM.

The life cycle starts with the system executive (an OS, an OS loader, and so forth) loading the MLE and SINIT AC module into available system memory. The system executive must validate and prepare the platform for the measured launch. When the platform is properly configured, the system executive executes GETSEC[SENTER] on the initiating logical processor (ILP) to rendezvous the responding logical processors into an SENTER sleep state, the ILP then enters into using the SINIT AC module. In a multi-threaded or multi-processing environment, the system executive must ensure that other logical processors are already in an idle loop, or asleep (such as after executing HLT) before executing GETSEC[SENTER].

After the GETSEC[SENTER] rendezvous handshake is performed between all logical processors in the platform, the ILP loads the chipset authenticated code module (SINIT) and performs an authentication check. If the check passes, the processor hashes the SINIT AC module and stores the result into TPM PCR 17. It then switches execution context to the SINIT AC module. The SINIT AC module will perform a number of platform operations, including: verifying the system configuration, protecting the system memory used by the MLE from I/O devices capable of DMA, producing a hash of the MLE, storing the hash value in TPM PCR 18, and various other operations. When SINIT completes execution, it executes the GETSEC[EXITAC] instruction and transfers control the MLE at the designated entry point.

Upon receiving control from the SINIT AC module, the MLE must establish its protection and isolation controls before enabling DMA and interrupts and transferring control to other software modules. It must also wake up the RLPs from their SENTER sleep state using the GETSEC[WAKEUP] instruction and bring them into its protection and isolation environment.

While executing in a measured environment, the MVMM can access the Trusted Platform Module (TPM) in locality 2. The MVMM has complete access to all TPM commands and may use the TPM to report current measurement values or use the measurement values to protect information such that only when the platform configuration registers (PCRs) contain the same value is the information released from the TPM. This protection mechanism is known as sealing.

A measured environment shutdown is ultimately completed by executing GETSEC[SEXIT]. Prior to this step system software is responsible for scrubbing sensitive information left in the processor caches, system memory.

7.3 GETSEC LEAF FUNCTIONS

This section provides detailed descriptions of each leaf function of the GETSEC instruction. GETSEC is available only if CPUID.01H:ECX[6] = 1. This indicates the availability of SMX and the GETSEC instruction. Before GETSEC can be executed, SMX must be enabled by setting CR4.SMXE[Bit 14] = 1.

A GETSEC leaf can only be used if it is shown to be available as reported by the GETSEC[CAPABILITIES] function. Attempts to access a GETSEC leaf index not supported by the processor, or if CR4.SMXE is 0, results in the signaling of an undefined opcode exception.

All GETSEC leaf functions are available in protected mode, including the compatibility sub-mode of IA-32e mode and the 64-bit sub-mode of IA-32e mode. Unless otherwise noted, the behavior of all GETSEC functions and interactions related to the measured environment are independent of IA-32e mode. This also applies to the interpretation of register widths[1] passed as input parameters to GETSEC functions and to register results returned as output parameters.

1. This chapter uses the 64-bit notation RAX, RIP, RSP, RFLAGS, etc. for processor registers because processors that support SMX also support Intel 64 Architecture. The MVMM can be launched in IA-32e mode or outside IA-32e mode. The 64-bit notation of processor registers also refer to its 32-bit forms if SMX is used in 32-bit environment. In some places, notation such as EAX is used to refer specifically to lower 32 bits of the indicated register.

The GETSEC functions ENTERACCS, SENTER, SEXIT, and WAKEUP require a Intel® TXT capable-chipset to be present in the platform. The GETSEC[CAPABILITIES] returned bit vector in position 0 indicates an Intel® TXT-capable chipset has been sampled present[1] by the processor.

The processor's operating mode also affects the execution of the following GETSEC leaf functions: SMCTRL, ENTERACCS, EXITAC, SENTER, SEXIT, and WAKEUP. These functions are only allowed in protected mode at CPL = 0. They are not allowed while in SMM in order to prevent potential intra-mode conflicts. Further execution qualifications exist to prevent potential architectural conflicts (for example: nesting of the measured environment or authenticated code execution mode). See the definitions of the GETSEC leaf functions for specific requirements.

For the purpose of performance monitor counting, the execution of GETSEC functions is counted as a single instruction with respect to retired instructions. The response by a responding logical processor (RLP) to messages associated with GETSEC[SENTER] or GTSEC[SEXIT] is transparent to the retired instruction count on the ILP.

1. Sampled present means that the processor sent a message to the chipset and the chipset responded that it (a) knows about the message and (b) is capable of executing SENTER. This means that the chipset CAN support Intel® TXT, and is configured and WILLING to support it.

GETSEC[CAPABILITIES]—Report the SMX Capabilities

Opcode	Instruction	Description
NP 0F 37 (EAX = 0)	GETSEC[CAPABILITIES]	Report the SMX capabilities. The capabilities index is input in EBX with the result returned in EAX.

Description

The GETSEC[CAPABILITIES] function returns a bit vector of supported GETSEC leaf functions. The CAPABILITIES leaf of GETSEC is selected with EAX set to 0 at entry. EBX is used as the selector for returning the bit vector field in EAX. GETSEC[CAPABILITIES] may be executed at all privilege levels, but the CR4.SMXE bit must be set or an undefined opcode exception (#UD) is returned.

With EBX = 0 upon execution of GETSEC[CAPABILITIES], EAX returns the a bit vector representing status on the presence of a Intel® TXT-capable chipset and the first 30 available GETSEC leaf functions. The format of the returned bit vector is provided in Table 7-3.

If bit 0 is set to 1, then an Intel® TXT-capable chipset has been sampled present by the processor. If bits in the range of 1-30 are set, then the corresponding GETSEC leaf function is available. If the bit value at a given bit index is 0, then the GETSEC leaf function corresponding to that index is unsupported and attempted execution results in a #UD.

Bit 31 of EAX indicates if further leaf indexes are supported. If the Extended Leafs bit 31 is set, then additional leaf functions are accessed by repeating GETSEC[CAPABILITIES] with EBX incremented by one. When the most significant bit of EAX is not set, then additional GETSEC leaf functions are not supported; indexing EBX to a higher value results in EAX returning zero.

Table 7-3. GETSEC Capability Result Encoding (EBX = 0)

Field	Bit position	Description
Chipset Present	0	Intel® TXT-capable chipset is present.
Undefined	1	Reserved
ENTERACCS	2	GETSEC[ENTERACCS] is available.
EXITAC	3	GETSEC[EXITAC] is available.
SENTER	4	GETSEC[SENTER] is available.
SEXIT	5	GETSEC[SEXIT] is available.
PARAMETERS	6	GETSEC[PARAMETERS] is available.
SMCTRL	7	GETSEC[SMCTRL] is available.
WAKEUP	8	GETSEC[WAKEUP] is available.
Undefined	30:9	Reserved
Extended Leafs	31	Reserved for extended information reporting of GETSEC capabilities.

Operation

```
IF (CR4.SMXE=0)
    THEN #UD;
ELSIF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
IF (EBX=0) THEN
        BitVector := 0;
        IF (TXT chipset present)
            BitVector[Chipset present] := 1;
        IF (ENTERACCS Available)
            THEN BitVector[ENTERACCS] := 1;
        IF (EXITAC Available)
            THEN BitVector[EXITAC] := 1;
        IF (SENTER Available)
            THEN BitVector[SENTER] := 1;
        IF (SEXIT Available)
            THEN BitVector[SEXIT] := 1;
        IF (PARAMETERS Available)
            THEN BitVector[PARAMETERS] := 1;
        IF (SMCTRL Available)
            THEN BitVector[SMCTRL] := 1;
        IF (WAKEUP Available)
            THEN BitVector[WAKEUP] := 1;
        EAX := BitVector;
ELSE
    EAX := 0;
END;;
```

Flags Affected

None.

Use of Prefixes

LOCK	Causes #UD.
REP*	Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
Operand size	Causes #UD.
NP	66/F2/F3 prefixes are not allowed.
Segment overrides	Ignored.
Address size	Ignored.
REX	Ignored.

Protected Mode Exceptions

#UD	If CR4.SMXE = 0.

Real-Address Mode Exceptions

#UD	If CR4.SMXE = 0.

Virtual-8086 Mode Exceptions

#UD	If CR4.SMXE = 0.

Compatibility Mode Exceptions

#UD	If CR4.SMXE = 0.

64-Bit Mode Exceptions

#UD	If CR4.SMXE = 0.

VM-exit Condition

Reason (GETSEC)	If in VMX non-root operation.

GETSEC[ENTERACCS]—Execute Authenticated Chipset Code

Opcode	Instruction	Description
NP 0F 37 (EAX = 2)	GETSEC[ENTERACCS]	Enter authenticated code execution mode. EBX holds the authenticated code module physical base address. ECX holds the authenticated code module size (bytes).

Description

The GETSEC[ENTERACCS] function loads, authenticates, and executes an authenticated code module using an Intel® TXT platform chipset's public key. The ENTERACCS leaf of GETSEC is selected with EAX set to 2 at entry.

There are certain restrictions enforced by the processor for the execution of the GETSEC[ENTERACCS] instruction:

- Execution is not allowed unless the processor is in protected mode or IA-32e mode with CPL = 0 and EFLAGS.VM = 0.
- Processor cache must be available and not disabled, that is, CR0.CD and CR0.NW bits must be 0.
- For processor packages containing more than one logical processor, CR0.CD is checked to ensure consistency between enabled logical processors.
- For enforcing consistency of operation with numeric exception reporting using Interrupt 16, CR0.NE must be set.
- An Intel TXT-capable chipset must be present as communicated to the processor by sampling of the power-on configuration capability field after reset.
- The processor can not already be in authenticated code execution mode as launched by a previous GETSEC[ENTERACCS] or GETSEC[SENTER] instruction without a subsequent exiting using GETSEC[EXITAC]).
- To avoid potential operability conflicts between modes, the processor is not allowed to execute this instruction if it currently is in SMM or VMX operation.
- To ensure consistent handling of SIPI messages, the processor executing the GETSEC[ENTERACCS] instruction must also be designated the BSP (boot-strap processor) as defined by IA32_APIC_BASE.BSP (Bit 8).

Failure to conform to the above conditions results in the processor signaling a general protection exception.

Prior to execution of the ENTERACCS leaf, other logical processors, i.e., RLPs, in the platform must be:

- Idle in a wait-for-SIPI state (as initiated by an INIT assertion or through reset for non-BSP designated processors), or
- In the SENTER sleep state as initiated by a GETSEC[SENTER] from the initiating logical processor (ILP).

If other logical processor(s) in the same package are not idle in one of these states, execution of ENTERACCS signals a general protection exception. The same requirement and action applies if the other logical processor(s) of the same package do not have CR0.CD = 0.

A successful execution of ENTERACCS results in the ILP entering an authenticated code execution mode. Prior to reaching this point, the processor performs several checks. These include:

- Establish and check the location and size of the specified authenticated code module to be executed by the processor.
- Inhibit the ILP's response to the external events: INIT, A20M, NMI, and SMI.
- Broadcast a message to enable protection of memory and I/O from other processor agents.
- Load the designated code module into an authenticated code execution area.
- Isolate the contents of the authenticated code execution area from further state modification by external agents.
- Authenticate the authenticated code module.
- Initialize the initiating logical processor state based on information contained in the authenticated code module header.
- Unlock the Intel® TXT-capable chipset private configuration space and TPM locality 3 space.

- Begin execution in the authenticated code module at the defined entry point.

The GETSEC[ENTERACCS] function requires two additional input parameters in the general purpose registers EBX and ECX. EBX holds the authenticated code (AC) module physical base address (the AC module must reside below 4 GBytes in physical address space) and ECX holds the AC module size (in bytes). The physical base address and size are used to retrieve the code module from system memory and load it into the internal authenticated code execution area. The base physical address is checked to verify it is on a modulo-4096 byte boundary. The size is verified to be a multiple of 64, that it does not exceed the internal authenticated code execution area capacity (as reported by GETSEC[CAPABILITIES]), and that the top address of the AC module does not exceed 32 bits. An error condition results in an abort of the authenticated code execution launch and the signaling of a general protection exception.

As an integrity check for proper processor hardware operation, execution of GETSEC[ENTERACCS] will also check the contents of all the machine check status registers (as reported by the MSRs IA32_MCi_STATUS) for any valid uncorrectable error condition. In addition, the global machine check status register IA32_MCG_STATUS MCIP bit must be cleared and the IERR processor package pin (or its equivalent) must not be asserted, indicating that no machine check exception processing is currently in progress. These checks are performed prior to initiating the load of the authenticated code module. Any outstanding valid uncorrectable machine check error condition present in these status registers at this point will result in the processor signaling a general protection violation.

The ILP masks the response to the assertion of the external signals INIT#, A20M, NMI#, and SMI#. This masking remains active until optionally unmasked by GETSEC[EXITAC] (this defined unmasking behavior assumes GETSEC[ENTERACCS] was not executed by a prior GETSEC[SENTER]). The purpose of this masking control is to prevent exposure to existing external event handlers that may not be under the control of the authenticated code module.

The ILP sets an internal flag to indicate it has entered authenticated code execution mode. The state of the A20M pin is likewise masked and forced internally to a de-asserted state so that any external assertion is not recognized during authenticated code execution mode.

To prevent other (logical) processors from interfering with the ILP operating in authenticated code execution mode, memory (excluding implicit write-back transactions) access and I/O originating from other processor agents are blocked. This protection starts when the ILP enters into authenticated code execution mode. Only memory and I/O transactions initiated from the ILP are allowed to proceed. Exiting authenticated code execution mode is done by executing GETSEC[EXITAC]. The protection of memory and I/O activities remains in effect until the ILP executes GETSEC[EXITAC].

Prior to launching the authenticated execution module using GETSEC[ENTERACCS] or GETSEC[SENTER], the processor's MTRRs (Memory Type Range Registers) must first be initialized to map out the authenticated RAM addresses as WB (writeback). Failure to do so may affect the ability for the processor to maintain isolation of the loaded authenticated code module. If the processor detected this requirement is not met, it will signal an Intel® TXT reset condition with an error code during the loading of the authenticated code module.

While physical addresses within the load module must be mapped as WB, the memory type for locations outside of the module boundaries must be mapped to one of the supported memory types as returned by GETSEC[PARAMETERS] (or UC as default).

To conform to the minimum granularity of MTRR MSRs for specifying the memory type, authenticated code RAM (ACRAM) is allocated to the processor in 4096 byte granular blocks. If an AC module size as specified in ECX is not a multiple of 4096 then the processor will allocate up to the next 4096 byte boundary for mapping as ACRAM with indeterminate data. This pad area will not be visible to the authenticated code module as external memory nor can it depend on the value of the data used to fill the pad area.

At the successful completion of GETSEC[ENTERACCS], the architectural state of the processor is partially initialized from contents held in the header of the authenticated code module. The processor GDTR, CS, and DS selectors are initialized from fields within the authenticated code module. Since the authenticated code module must be relocatable, all address references must be relative to the authenticated code module base address in EBX. The processor GDTR base value is initialized to the AC module header field GDTBasePtr + module base address held in EBX and the GDTR limit is set to the value in the GDTLimit field. The CS selector is initialized to the AC module header SegSel field, while the DS selector is initialized to CS + 8. The segment descriptor fields are implicitly initialized to BASE=0, LIMIT=FFFFFh, G=1, D=1, P=1, S=1, read/write access for DS, and execute/read access for CS. The processor begins the authenticated code module execution with the EIP set to the AC module header EntryPoint field + module base address (EBX). The AC module based fields used for initializing the processor state are checked for consistency and any failure results in a shutdown condition.

A summary of the register state initialization after successful completion of GETSEC[ENTERACCS] is given for the processor in Table 7-4. The paging is disabled upon entry into authenticated code execution mode. The authenticated code module is loaded and initially executed using physical addresses. It is up to the system software after execution of GETSEC[ENTERACCS] to establish a new (or restore its previous) paging environment with an appropriate mapping to meet new protection requirements. EBP is initialized to the authenticated code module base physical address for initial execution in the authenticated environment. As a result, the authenticated code can reference EBP for relative address based references, given that the authenticated code module must be position independent.

Table 7-4. Register State Initialization After GETSEC[ENTERACCS]

Register State	Initialization Status	Comment
CR0	PG←0, AM←0, WP←0: Others unchanged	Paging, Alignment Check, Write-protection are disabled.
CR4	MCE←0, CET←0, PCIDE←0, FRED←0: Others unchanged	Machine Check Exceptions, Control-flow Enforcement Technology, Process-context Identifiers, and FRED disabled.
EFLAGS	00000002H	
IA32_EFER	0H	IA-32e mode disabled.
EIP	AC.base + EntryPoint	AC.base is in EBX as input to GETSEC[ENTERACCS].
[E|R]BX	Pre-ENTERACCS state: Next [E|R]IP prior to GETSEC[ENTERACCS]	Carry forward 64-bit processor state across GETSEC[ENTERACCS].
ECX	Pre-ENTERACCS state: [31:16]=GDTR.limit; [15:0]=CS.sel	Carry forward processor state across GETSEC[ENTERACCS].
[E|R]DX	Pre-ENTERACCS state: GDTR base	Carry forward 64-bit processor state across GETSEC[ENTERACCS].
EBP	AC.base	
CS	Sel=[SegSel], base=0, limit=FFFFFh, G=1, D=1, AR=9BH	
DS	Sel=[SegSel] +8, base=0, limit=FFFFFh, G=1, D=1, AR=93H	
GDTR	Base= AC.base (EBX) + [GDTBasePtr], Limit=[GDTLimit]	
DR7	00000400H	
IA32_DEBUGCTL	0H	
IA32_MISC_ENABLE	See Table 7-5 for example.	The number of initialized fields may change due to processor implementation.
Performance counters and counter control registers	0H	

The segmentation related processor state that has not been initialized by GETSEC[ENTERACCS] requires appropriate initialization before use. Since a new GDT context has been established, the previous state of the segment selector values held in ES, SS, FS, GS, TR, and LDTR might not be valid.

The MSR IA32_EFER is also unconditionally cleared as part of the processor state initialized by ENTERACCS. Since paging is disabled upon entering authenticated code execution mode, a new paging environment will have to be reestablished in order to establish IA-32e mode while operating in authenticated code execution mode.

Debug exception and trap related signaling is also disabled as part of GETSEC[ENTERACCS]. This is achieved by resetting DR7, TF in EFLAGs, and the MSR IA32_DEBUGCTL. These debug functions are free to be re-enabled once supporting exception handler(s), descriptor tables, and debug registers have been properly initialized following entry into authenticated code execution mode. Also, any pending single-step trap condition will have been cleared upon entry into this mode.

Performance related counters and counter control registers are cleared as part of execution of ENTERACCS. This implies any active performance counters at any time of ENTERACCS execution will be disabled. To reactive the processor performance counters, this state must be re-initialized and re-enabled.

The IA32_MISC_ENABLE MSR is initialized upon entry into authenticated execution mode. Certain bits of this MSR are preserved because preserving these bits may be important to maintain previously established platform settings (See the footnote for Table 7-5.). The remaining bits are cleared for the purpose of establishing a more consistent environment for the execution of authenticated code modules. One of the impacts of initializing this MSR is any previous condition established by the MONITOR instruction will be cleared.

To support the possible return to the processor architectural state prior to execution of GETSEC[ENTERACCS], certain critical processor state is captured and stored in the general- purpose registers at instruction completion. [E|R]BX holds effective address ([E|R]IP) of the instruction that would execute next after GETSEC[ENTERACCS], ECX[15:0] holds the CS selector value, ECX[31:16] holds the GDTR limit field, and [E|R]DX holds the GDTR base field. The subsequent authenticated code can preserve the contents of these registers so that this state can be manually restored if needed, prior to exiting authenticated code execution mode with GETSEC[EXITAC]. For the processor state after exiting authenticated code execution mode, see the description of GETSEC[SEXIT].

Table 7-5. IA32_MISC_ENABLE MSR Initialization[1] by ENTERACCS and SENTER

Field	Bit position	Description
Fast strings enable	0	Clear to 0.
FOPCODE compatibility mode enable	2	Clear to 0.
Thermal monitor enable	3	Set to 1 if other thermal monitor capability is not enabled.[2]
Split-lock disable	4	Clear to 0.
Bus lock on cache line splits disable	8	Clear to 0.
Hardware prefetch disable	9	Clear to 0.
GV1/2 legacy enable	15	Clear to 0.
MONITOR/MWAIT s/m enable	18	Clear to 0.
Adjacent sector prefetch disable	19	Clear to 0.

NOTES:

1. The number of IA32_MISC_ENABLE fields that are initialized may vary due to processor implementations.
2. ENTERACCS (and SENTER) initialize the state of processor thermal throttling such that at least a minimum level is enabled. If thermal throttling is already enabled when executing one of these GETSEC leaves, then no change in the thermal throttling control settings will occur. If thermal throttling is disabled, then it will be enabled via setting of the thermal throttle control bit 3 as a result of executing these GETSEC leaves.

The IDTR will also require reloading with a new IDT context after entering authenticated code execution mode, before any exceptions or the external interrupts INTR and NMI can be handled. Since external interrupts are re-enabled at the completion of authenticated code execution mode (as terminated with EXITAC), it is recommended that a new IDT context be established before this point. Until such a new IDT context is established, the programmer must take care in not executing an INT n instruction or any other operation that would result in an exception or trap signaling.

Prior to completion of the GETSEC[ENTERACCS] instruction and after successful authentication of the AC module, the private configuration space of the Intel TXT chipset is unlocked. The authenticated code module alone can gain access to this normally restricted chipset state for the purpose of securing the platform.

Once the authenticated code module is launched at the completion of GETSEC[ENTERACCS], it is free to enable interrupts by setting EFLAGS.IF and enable NMI by execution of IRET. This presumes that it has re-established interrupt handling support through initialization of the IDT, GDT, and corresponding interrupt handling code.

Operation in a Uni-Processor Platform

```
(* The state of the internal flag ACMODEFLAG persists across instruction boundary *)
IF (CR4.SMXE=0)
    THEN #UD;
ELSIF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSIF (GETSEC leaf unsupported)
    THEN #UD;
ELSIF ((in VMX operation) or
    (CR0.PE=0) or (CR0.CD=1) or (CR0.NW=1) or (CR0.NE=0) or
    (CPL>0) or (EFLAGS.VM=1) or
    (IA32_APIC_BASE.BSP=0) or
    (TXT chipset not present) or
    (ACMODEFLAG=1) or (IN_SMM=1))
        THEN #GP(0);
IF (GETSEC[PARAMETERS].Parameter_Type = 5, MCA_Handling (bit 6) = 0)
    FOR I = 0 to IA32_MCG_CAP.COUNT-1 DO
        IF (IA32_MC[I]_STATUS = uncorrectable error)
            THEN #GP(0);
    OD;
FI;
IF (IA32_MCG_STATUS.MCIP=1) or (IERR pin is asserted)
    THEN #GP(0);
ACBASE := EBX;
ACSIZE := ECX;
IF (((ACBASE MOD 4096) ≠ 0) or ((ACSIZE MOD 64 ) ≠ 0 ) or (ACSIZE < minimum module size) OR (ACSIZE > authenticated RAM
capacity)) or ((ACBASE+ACSIZE) > (2^32 -1)))
    THEN #GP(0);
IF (secondary thread(s) CR0.CD = 1) or ((secondary thread(s) NOT(wait-for-SIPI)) and
    (secondary thread(s) not in SENTER sleep state)
    THEN #GP(0);
Mask SMI, INIT, A20M, and NMI external pin events;
IA32_MISC_ENABLE := (IA32_MISC_ENABLE & MASK_CONST*)
(* The hexadecimal value of MASK_CONST may vary due to processor implementations *)
A20M := 0;
IA32_DEBUGCTL := 0;
Invalidate processor TLB(s);
Drain Outgoing Transactions;
ACMODEFLAG := 1;
SignalTXTMessage(ProcessorHold);
Load the internal ACRAM based on the AC module size;
(* Ensure that all ACRAM loads hit Write Back memory space *)
IF (ACRAM memory type ≠ WB)
    THEN TXT-SHUTDOWN(#BadACMMType);
IF (AC module header version isnot supported) OR (ACRAM[ModuleType] ≠ 2)
    THEN TXT-SHUTDOWN(#UnsupportedACM);
 (* Authenticate the AC Module and shutdown with an error if it fails *)
KEY := GETKEY(ACRAM, ACBASE);
KEYHASH := HASH(KEY);
CSKEYHASH := READ(TXT.PUBLIC.KEY);
IF (KEYHASH ≠ CSKEYHASH)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
SIGNATURE := DECRYPT(ACRAM, ACBASE, KEY);
(* The value of SIGNATURE_LEN_CONST is implementation-specific*)
```

```
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.I] := SIGNATURE[I];
COMPUTEDSIGNATURE := HASH(ACRAM, ACBASE, ACSIZE);
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.SIGNATURE_LEN_CONST+I] := COMPUTEDSIGNATURE[I];
IF (SIGNATURE ≠ COMPUTEDSIGNATURE)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
ACMCONTROL := ACRAM[CodeControl];
IF ((ACMCONTROL.0 = 0) and (ACMCONTROL.1 = 1) and (snoop hit to modified line detected on ACRAM load))
    THEN TXT-SHUTDOWN(#UnexpectedHITM);
IF (ACMCONTROL reserved bits are set)
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[GDTBasePtr] < (ACRAM[HeaderLen] * 4 + Scratch_size)) OR
    ((ACRAM[GDTBasePtr] + ACRAM[GDTLimit]) >= ACSIZE))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACMCONTROL.0 = 1) and (ACMCONTROL.1 = 1) and (snoop hit to modified line detected on ACRAM load))
    THEN ACEntryPoint := ACBASE+ACRAM[ErrorEntryPoint];
ELSE
    ACEntryPoint := ACBASE+ACRAM[EntryPoint];
IF ((ACEntryPoint >= ACSIZE) OR (ACEntryPoint < (ACRAM[HeaderLen] * 4 + Scratch_size)))THEN TXT-SHUTDOWN(#BadACMFormat);
IF (ACRAM[GDTLimit] & FFFF0000h)
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel] > (ACRAM[GDTLimit] - 15)) OR (ACRAM[SegSel] < 8))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel].TI=1) OR (ACRAM[SegSel].RPL≠0))
    THEN TXT-SHUTDOWN(#BadACMFormat);
CR0.[PG.AM.WP] := 0;
CR4.MCE := 0;
ACRAM[CR4High].FRED := CR4.FRED;
CR4.FRED := 0;
EFLAGS := 00000002h;
IA32_EFER := 0h;
[E|R]BX := [E|R]IP of the instruction after GETSEC[ENTERACCS];
ECX := Pre-GETSEC[ENTERACCS] GDT.limit:CS.sel;
[E|R]DX := Pre-GETSEC[ENTERACCS] GDT.base;
EBP := ACBASE;
GDTR.BASE := ACBASE+ACRAM[GDTBasePtr];
GDTR.LIMIT := ACRAM[GDTLimit];
CS.SEL := ACRAM[SegSel];
CS.BASE := 0;
CS.LIMIT := FFFFFh;
CS.G := 1;
CS.D := 1;
CS.AR := 9Bh;
DS.SEL := ACRAM[SegSel]+8;
DS.BASE := 0;
DS.LIMIT := FFFFFh;
DS.G := 1;
DS.D := 1;
DS.AR := 93h;
DR7 := 00000400h;
IA32_DEBUGCTL := 0;
SignalTXTMsg(OpenPrivate);
SignalTXTMsg(OpenLocality3);
```

```
EIP := ACEntryPoint;
END;
```

Flags Affected

All flags are cleared.

Use of Prefixes

LOCK	Causes #UD.
REP*	Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
Operand size	Causes #UD.
NP	66/F2/F3 prefixes are not allowed.
Segment overrides	Ignored.
Address size	Ignored.
REX	Ignored.

Protected Mode Exceptions

#UD	If CR4.SMXE = 0.
	If GETSEC[ENTERACCS] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	If CR0.CD = 1 or CR0.NW = 1 or CR0.NE = 0 or CR0.PE = 0 or CPL > 0 or EFLAGS.VM = 1.
	If a Intel® TXT-capable chipset is not present.
	If in VMX root operation.
	If the initiating processor is not designated as the bootstrap processor via the MSR bit IA32_APIC_BASE.BSP.
	If the processor is already in authenticated code execution mode.
	If the processor is in SMM.
	If a valid uncorrectable machine check error is logged in IA32_MC[I]_STATUS.
	If the authenticated code base is not on a 4096 byte boundary.
	If the authenticated code size > processor internal authenticated code area capacity.
	If the authenticated code size is not modulo 64.
	If other enabled logical processor(s) of the same package CR0.CD = 1.
	If other enabled logical processor(s) of the same package are not in the wait-for-SIPI or SENTER sleep state.

Real-Address Mode Exceptions

#UD	If CR4.SMXE = 0.
	If GETSEC[ENTERACCS] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	GETSEC[ENTERACCS] is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	If CR4.SMXE = 0. If GETSEC[ENTERACCS] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	GETSEC[ENTERACCS] is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

All protected mode exceptions apply.

#GP	If AC code module does not reside in physical address below 2^32 -1.

64-Bit Mode Exceptions

All protected mode exceptions apply.

#GP	If AC code module does not reside in physical address below 2^32 -1.

VM-exit Condition

Reason (GETSEC)	If in VMX non-root operation.

GETSEC[EXITAC]—Exit Authenticated Code Execution Mode

Opcode	Instruction	Description
NP 0F 37 (EAX=3)	GETSEC[EXITAC]	Exit authenticated code execution mode. RBX holds the Near Absolute Indirect jump target and EDX hold the exit parameter flags.

Description

The GETSEC[EXITAC] leaf function exits the ILP out of authenticated code execution mode established by GETSEC[ENTERACCS] or GETSEC[SENTER]. The EXITAC leaf of GETSEC is selected with EAX set to 3 at entry. EBX (or RBX, if in 64-bit mode) holds the near jump target offset for where the processor execution resumes upon exiting authenticated code execution mode. EDX contains additional parameter control information. Currently only an input value of 0 in EDX is supported. All other EDX settings are considered reserved and result in a general protection violation.

GETSEC[EXITAC] can only be executed if the processor is in protected mode with CPL = 0 and EFLAGS.VM = 0. The processor must also be in authenticated code execution mode. To avoid potential operability conflicts between modes, the processor is not allowed to execute this instruction if it is in SMM or in VMX operation. A violation of these conditions results in a general protection violation.

Upon completion of the GETSEC[EXITAC] operation, the processor unmasks responses to external event signals INIT#, NMI#, and SMI#. This unmasking is performed conditionally, based on whether the authenticated code execution mode was entered via execution of GETSEC[SENTER] or GETSEC[ENTERACCS]. If the processor is in authenticated code execution mode due to the execution of GETSEC[SENTER], then these external event signals will remain masked. In this case, A20M is kept disabled in the measured environment until the measured environment executes GETSEC[SEXIT]. INIT# is unconditionally unmasked by EXITAC. Note that any events that are pending, but have been blocked while in authenticated code execution mode, will be recognized at the completion of the GETSEC[EXITAC] instruction if the pin event is unmasked.

The intent of providing the ability to optionally leave the pin events SMI#, and NMI# masked is to support the completion of a measured environment bring-up that makes use of VMX. In this envisioned security usage scenario, these events will remain masked until an appropriate virtual machine has been established in order to field servicing of these events in a safer manner. Details on when and how events are masked and unmasked in VMX operation are described in Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C. It should be cautioned that if no VMX environment is to be activated following GETSEC[EXITAC], that these events will remain masked until the measured environment is exited with GETSEC[SEXIT]. If this is not desired then the GETSEC function SMCTRL(0) can be used for unmasking SMI# in this context. NMI# can be correspondingly unmasked by execution of IRET.

A successful exit of the authenticated code execution mode requires the ILP to perform additional steps as outlined below:

- Invalidate the contents of the internal authenticated code execution area.
- Invalidate processor TLBs.
- Clear the internal processor AC Mode indicator flag.
- Re-lock the TPM locality 3 space.
- Unlock the Intel® TXT-capable chipset memory and I/O protections to allow memory and I/O activity by other processor agents.
- Perform a near absolute indirect jump to the designated instruction location.

The content of the authenticated code execution area is invalidated by hardware in order to protect it from further use or visibility. This internal processor storage area can no longer be used or relied upon after GETSEC[EXITAC]. Data structures need to be re-established outside of the authenticated code execution area if they are to be referenced after EXITAC. Since addressed memory content formerly mapped to the authenticated code execution area may no longer be coherent with external system memory after EXITAC, processor TLBs in support of linear to physical address translation are also invalidated.

Upon completion of GETSEC[EXITAC] a near absolute indirect transfer is performed with EIP loaded with the contents of EBX (based on the current operating mode size). In 64-bit mode, all 64 bits of RBX are loaded into RIP if REX.W precedes GETSEC[EXITAC]. Otherwise RBX is treated as 32 bits even while in 64-bit mode. Conventional CS limit checking is performed as part of this control transfer. Any exception conditions generated as part of this control transfer will be directed to the existing IDT; thus it is recommended that an IDTR should also be established prior to execution of the EXITAC function if there is a need for fault handling. In addition, any segmentation related (and paging) data structures to be used after EXITAC should be re-established or validated by the authenticated code prior to EXITAC.

In addition, any segmentation related (and paging) data structures to be used after EXITAC need to be re-established and mapped outside of the authenticated RAM designated area by the authenticated code prior to EXITAC. Any data structure held within the authenticated RAM allocated area will no longer be accessible after completion by EXITAC.

Operation

```
(* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary *)
IF (CR4.SMXE=0)
    THEN #UD;
ELSIF ( in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSIF (GETSEC leaf unsupported)
    THEN #UD;
ELSIF ((in VMX operation) or ( (in 64-bit mode) and ( RBX is non-canonical) )
    (CR0.PE=0) or (CPL>0) or (EFLAGS.VM=1) or
    (ACMODEFLAG=0) or (IN_SMM=1)) or (EDX ≠ 0))
    THEN #GP(0);
IF (OperandSize = 32)
    THEN tempEIP := EBX;
ELSIF (OperandSize = 64)
    THEN tempEIP := RBX;
ELSE
    tempEIP := EBX AND 0000FFFFH;
IF (tempEIP > code segment limit)
    THEN #GP(0);
IF (ACRAM[CR4High].FRED = 1) and (IA32_EFER.LMA = 0)
    THEN #GP(0);
ELSE CR4.FRED = ACRAM[CR4High].FRED;
Invalidate ACRAM contents;
Invalidate processor TLB(s);
Drain outgoing messages;
SignalTXTMsg(CloseLocality3);
SignalTXTMsg(LockSMRAM);
SignalTXTMsg(ProcessorRelease);
Unmask INIT;
IF (SENTERFLAG=0)
    THEN Unmask SMI, INIT, NMI, and A20M pin event;
ELSEIF (IA32_SMM_MONITOR_CTL[0] = 0)
    THEN Unmask SMI pin event;
ACMODEFLAG := 0;
IF IA32_EFER.LMA == 1
    THEN CR3 := R8;
EIP := tempEIP;
END;
```

Flags Affected

None.

Use of Prefixes

LOCK	Causes #UD.
REP*	Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
Operand size	Causes #UD.
NP	66/F2/F3 prefixes are not allowed.
Segment overrides	Ignored.
Address size	Ignored.
REX.W	Sets 64-bit mode Operand size attribute.

Protected Mode Exceptions

#UD	If CR4.SMXE = 0.
	If GETSEC[EXITAC] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	If CR0.PE = 0 or CPL>0 or EFLAGS.VM =1.
	If in VMX root operation.
	If the processor is not currently in authenticated code execution mode.
	If the processor is in SMM.
	If any reserved bit position is set in the EDX parameter register.

Real-Address Mode Exceptions

#UD	If CR4.SMXE = 0.
	If GETSEC[EXITAC] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	GETSEC[EXITAC] is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	If CR4.SMXE = 0.
	If GETSEC[EXITAC] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	GETSEC[EXITAC] is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

All protected mode exceptions apply.

64-Bit Mode Exceptions

All protected mode exceptions apply.

#GP(0)	If the target address in RBX is not in a canonical form.

VM-Exit Condition

Reason (GETSEC)	If in VMX non-root operation.

GETSEC[SENTER]—Enter a Measured Environment

Opcode	Instruction	Description
NP 0F 37 (EAX=4)	GETSEC[SENTER]	Launch a measured environment. EBX holds the SINIT authenticated code module physical base address. ECX holds the SINIT authenticated code module size (bytes). EDX controls the level of functionality supported by the measured environment launch.

Description

The GETSEC[SENTER] instruction initiates the launch of a measured environment and places the initiating logical processor (ILP) into the authenticated code execution mode. The SENTER leaf of GETSEC is selected with EAX set to 4 at execution. The physical base address of the AC module to be loaded and authenticated is specified in EBX. The size of the module in bytes is specified in ECX. EDX controls the level of functionality supported by the measured environment launch. To enable the full functionality of the protected environment launch, EDX must be initialized to zero.

The authenticated code base address and size parameters (in bytes) are passed to the GETSEC[SENTER] instruction using EBX and ECX respectively. The ILP evaluates the contents of these registers according to the rules for the AC module address in GETSEC[ENTERACCS]. AC module execution follows the same rules, as set by GETSEC[ENTERACCS].

The launching software must ensure that the TPM.ACCESS_0.activeLocality bit is clear before executing the GETSEC[SENTER] instruction.

There are restrictions enforced by the processor for execution of the GETSEC[SENTER] instruction:

- Execution is not allowed unless the processor is in protected mode or IA-32e mode with CPL = 0 and EFLAGS.VM = 0.
- Processor cache must be available and not disabled using the CR0.CD and NW bits.
- For enforcing consistency of operation with numeric exception reporting using Interrupt 16, CR0.NE must be set.
- An Intel TXT-capable chipset must be present as communicated to the processor by sampling of the power-on configuration capability field after reset.
- The processor can not be in authenticated code execution mode or already in a measured environment (as launched by a previous GETSEC[ENTERACCS] or GETSEC[SENTER] instruction).
- To avoid potential operability conflicts between modes, the processor is not allowed to execute this instruction if it currently is in SMM or VMX operation.
- To ensure consistent handling of SIPI messages, the processor executing the GETSEC[SENTER] instruction must also be designated the BSP (boot-strap processor) as defined by IA32_APIC_BASE.BSP (Bit 8).
- EDX must be initialized to a setting supportable by the processor. Unless enumeration by the GETSEC[PARAMETERS] leaf reports otherwise, only a value of zero is supported.

Failure to abide by the above conditions results in the processor signaling a general protection violation.

This instruction leaf starts the launch of a measured environment by initiating a rendezvous sequence for all logical processors in the platform. The rendezvous sequence involves the initiating logical processor sending a message (by executing GETSEC[SENTER]) and other responding logical processors (RLPs) acknowledging the message, thus synchronizing the RLP(s) with the ILP.

In response to a message signaling the completion of rendezvous, RLPs clear the bootstrap processor indicator flag (IA32_APIC_BASE.BSP) and enter an SENTER sleep state. In this sleep state, RLPs enter an idle processor condition while waiting to be activated after a measured environment has been established by the system executive. RLPs in the SENTER sleep state can only be activated by the GETSEC leaf function WAKEUP in a measured environment.

A successful launch of the measured environment results in the initiating logical processor entering the authenticated code execution mode. Prior to reaching this point, the ILP performs the following steps internally:

- Inhibit processor response to the external events: INIT, A20M, NMI, and SMI.
- Establish and check the location and size of the authenticated code module to be executed by the ILP.
- Check for the existence of an Intel® TXT-capable chipset.
- Verify the current power management configuration is acceptable.
- Broadcast a message to enable protection of memory and I/O from activities from other processor agents.
- Load the designated AC module into authenticated code execution area.
- Isolate the content of authenticated code execution area from further state modification by external agents.
- Authenticate the AC module.
- Updated the Trusted Platform Module (TPM) with the authenticated code module's hash.
- Initialize processor state based on the authenticated code module header information.
- Unlock the Intel® TXT-capable chipset private configuration register space and TPM locality 3 space.
- Begin execution in the authenticated code module at the defined entry point.

As an integrity check for proper processor hardware operation, execution of GETSEC[SENTER] will also check the contents of all the machine check status registers (as reported by the MSRs IA32_MCi_STATUS) for any valid uncorrectable error condition. In addition, the global machine check status register IA32_MCG_STATUS MCIP bit must be cleared and the IERR processor package pin (or its equivalent) must be not asserted, indicating that no machine check exception processing is currently in-progress. These checks are performed twice: once by the ILP prior to the broadcast of the rendezvous message to RLPs, and later in response to RLPs acknowledging the rendezvous message. Any outstanding valid uncorrectable machine check error condition present in the machine check status registers at the first check point will result in the ILP signaling a general protection violation. If an outstanding valid uncorrectable machine check error condition is present at the second check point, then this will result in the corresponding logical processor signaling the more severe TXT-shutdown condition with an error code of 12.

Before loading and authentication of the target code module is performed, the processor also checks that the current voltage and bus ratio encodings correspond to known good values supportable by the processor. The MSR IA32_PERF_STATUS values are compared against either the processor supported maximum operating target setting, system reset setting, or the thermal monitor operating target. If the current settings do not meet any of these criteria then the SENTER function will attempt to change the voltage and bus ratio select controls in a processor-specific manner. This adjustment may be to the thermal monitor, minimum (if different), or maximum operating target depending on the processor.

This implies that some thermal operating target parameters configured by BIOS may be overridden by SENTER. The measured environment software may need to take responsibility for restoring such settings that are deemed to be safe, but not necessarily recognized by SENTER. If an adjustment is not possible when an out of range setting is discovered, then the processor will abort the measured launch. This may be the case for chipset controlled settings of these values or if the controllability is not enabled on the processor. In this case it is the responsibility of the external software to program the chipset voltage ID and/or bus ratio select settings to known good values recognized by the processor, prior to executing SENTER.

NOTE

For a mobile processor, an adjustment can be made according to the thermal monitor operating target. For a quad-core processor the SENTER adjustment mechanism may result in a more conservative but non-uniform voltage setting, depending on the pre-SENTER settings per core.

The ILP and RLPs mask the response to the assertion of the external signals INIT#, A20M, NMI#, and SMI#. The purpose of this masking control is to prevent exposure to existing external event handlers until a protected handler has been put in place to directly handle these events. Masked external pin events may be unmasked conditionally or unconditionally via the GETSEC[EXITAC], GETSEC[SEXIT], GETSEC[SMCTRL] or for specific VMX related operations such as a VM entry or the VMXOFF instruction (see respective GETSEC leaves and Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C, for more details). The state of the A20M pin is masked and forced internally to a de-asserted state so that external assertion is not recognized. A20M masking as set by

GETSEC[SENTER] is undone only after taking down the measured environment with the GETSEC[SEXIT] instruction or processor reset. INTR is masked by simply clearing the EFLAGS.IF bit. It is the responsibility of system software to control the processor response to INTR through appropriate management of EFLAGS.

To prevent other (logical) processors from interfering with the ILP operating in authenticated code execution mode, memory (excluding implicit write-back transactions) and I/O activities originating from other processor agents are blocked. This protection starts when the ILP enters into authenticated code execution mode. Only memory and I/O transactions initiated from the ILP are allowed to proceed. Exiting authenticated code execution mode is done by executing GETSEC[EXITAC]. The protection of memory and I/O activities remains in effect until the ILP executes GETSEC[EXITAC].

Once the authenticated code module has been loaded into the authenticated code execution area, it is protected against further modification from external bus snoops. There is also a requirement that the memory type for the authenticated code module address range be WB (via initialization of the MTRRs prior to execution of this instruction). If this condition is not satisfied, it is a violation of security and the processor will force a TXT system reset (after writing an error code to the chipset LT.ERRORCODE register). This action is referred to as a Intel® TXT reset condition. It is performed when it is considered unreliable to signal an error through the conventional exception reporting mechanism.

To conform to the minimum granularity of MTRR MSRs for specifying the memory type, authenticated code RAM (ACRAM) is allocated to the processor in 4096 byte granular blocks. If an AC module size as specified in ECX is not a multiple of 4096 then the processor will allocate up to the next 4096 byte boundary for mapping as ACRAM with indeterminate data. This pad area will not be visible to the authenticated code module as external memory nor can it depend on the value of the data used to fill the pad area.

Once successful authentication has been completed by the ILP, the computed hash is stored in a trusted storage facility in the platform. The following trusted storage facility are supported:

- If the platform register FTM_INTERFACE_ID.[bits 3:0] = 0, the computed hash is stored to the platform's TPM at PCR17 after this register is implicitly reset. PCR17 is a dedicated register for holding the computed hash of the authenticated code module loaded and subsequently executed by the GETSEC[SENTER]. As part of this process, the dynamic PCRs 18-22 are reset so they can be utilized by subsequently software for registration of code and data modules.
- If the platform register FTM_INTERFACE_ID.[bits 3:0] = 1, the computed hash is stored in a firmware trusted module (FTM) using a modified protocol similar to the protocol used to write to TPM's PCR17.

After successful execution of SENTER, either PCR17 (if FTM is not enabled) or the FTM (if enabled) contains the measurement of AC code and the SENTER launching parameters.

After authentication is completed successfully, the private configuration space of the Intel® TXT-capable chipset is unlocked so that the authenticated code module and measured environment software can gain access to this normally restricted chipset state. The Intel® TXT-capable chipset private configuration space can be locked later by software writing to the chipset LT.CMD.CLOSE-PRIVATE register or unconditionally using the GETSEC[SEXIT] instruction.

The SENTER leaf function also initializes some processor architecture state for the ILP from contents held in the header of the authenticated code module. Since the authenticated code module is relocatable, all address references are relative to the base address passed in via EBX. The ILP GDTR base value is initialized to EBX + [GDTBasePtr] and GDTR limit set to [GDTLimit]. The CS selector is initialized to the value held in the AC module header field SegSel, while the DS, SS, and ES selectors are initialized to CS+8. The segment descriptor fields are initialized implicitly with BASE=0, LIMIT=FFFFFh, G=1, D=1, P=1, S=1, read/write/accessed for DS, SS, and ES, while execute/read/accessed for CS. Execution in the authenticated code module for the ILP begins with the EIP set to EBX + [EntryPoint]. AC module defined fields used for initializing processor state are consistency checked with a failure resulting in an TXT-shutdown condition.

Table 7-6 provides a summary of processor state initialization for the ILP and RLP(s) after successful completion of GETSEC[SENTER]. For both ILP and RLP(s), paging is disabled upon entry to the measured environment. It is up to the ILP to establish a trusted paging environment, with appropriate mappings, to meet protection requirements established during the launch of the measured environment. RLP state initialization is not completed until a subsequent wake-up has been signaled by execution of the GETSEC[WAKEUP] function by the ILP.

Table 7-6. Register State Initialization After GETSEC[SENTER] and GETSEC[WAKEUP]

Register State	ILP after GETSEC[SENTER]	RLP after GETSEC[WAKEUP]
CR0	PG←0, AM←0, WP←0; Others unchanged	PG←0, CD←0, NW←0, AM←0, WP←0; PE←1, NE←1
CR4	00004000H	00004000H
EFLAGS	00000002H	00000002H
IA32_EFER	0H	0
EIP	[EntryPoint from MLE header[1]]	[LT.MLE.JOIN + 12]
EBX	Unchanged [SINIT.BASE]	Unchanged
EDX	SENTER control flags	Unchanged
EBP	SINIT.BASE	Unchanged
CS	Sel=[SINIT SegSel], base=0, limit=FFFFFh, G=1, D=1, AR=9BH	Sel = [LT.MLE.JOIN + 8], base = 0, limit = FFFFFH, G = 1, D = 1, AR = 9BH
DS, ES, SS	Sel=[SINIT SegSel] +8, base=0, limit=FFFFFh, G=1, D=1, AR=93H	Sel = [LT.MLE.JOIN + 8] +8, base = 0, limit = FFFFFH, G = 1, D = 1, AR = 93H
GDTR	Base= SINIT.base (EBX) + [SINIT.GDTBasePtr], Limit=[SINIT.GDTLimit]	Base = [LT.MLE.JOIN + 4], Limit = [LT.MLE.JOIN]
DR7	00000400H	00000400H
IA32_DEBUGCTL	0H	0H
Performance counters and counter control registers	0H	0H
IA32_MISC_ENABLE	See Table 7-5	See Table 7-5
IA32_SMM_MONITOR_CTL	Bit 2←0	Bit 2←0

NOTES:

1. See the *Intel® Trusted Execution Technology Measured Launched Environment Programming Guide* for MLE header format.

Segmentation related processor state that has not been initialized by GETSEC[SENTER] requires appropriate initialization before use. Since a new GDT context has been established, the previous state of the segment selector values held in FS, GS, TR, and LDTR may no longer be valid. The IDTR will also require reloading with a new IDT context after launching the measured environment before exceptions or the external interrupts INTR and NMI can be handled. In the meantime, the programmer must take care in not executing an INT n instruction or any other condition that would result in an exception or trap signaling.

Debug exception and trap related signaling is also disabled as part of execution of GETSEC[SENTER]. This is achieved by clearing DR7, TF in EFLAGs, and the MSR IA32_DEBUGCTL as defined in Table 7-6. These can be re-enabled once supporting exception handler(s), descriptor tables, and debug registers have been properly re-initialized following SENTER. Also, any pending single-step trap condition will be cleared at the completion of SENTER for both the ILP and RLP(s).

Performance related counters and counter control registers are cleared as part of execution of SENTER on both the ILP and RLP. This implies any active performance counters at the time of SENTER execution will be disabled. To reactive the processor performance counters, this state must be re-initialized and re-enabled.

Since MCE along with all other state bits (with the exception of SMXE) are cleared in CR4 upon execution of SENTER processing, any enabled machine check error condition that occurs will result in the processor performing the TXT-shutdown action. This also applies to an RLP while in the SENTER sleep state. For each logical processor CR4.MCE must be reestablished with a valid machine check exception handler to otherwise avoid an TXT-shutdown under such conditions.

The MSR IA32_EFER is also unconditionally cleared as part of the processor state initialized by SENTER for both the ILP and RLP. Since paging is disabled upon entering authenticated code execution mode, a new paging environment will have to be re-established if it is desired to enable IA-32e mode while operating in authenticated code execution mode.

The miscellaneous feature control MSR, IA32_MISC_ENABLE, is initialized as part of the measured environment launch. Certain bits of this MSR are preserved because preserving these bits may be important to maintain previously established platform settings. See the footnote for Table 7-5 The remaining bits are cleared for the purpose of establishing a more consistent environment for the execution of authenticated code modules. Among the impact of initializing this MSR, any previous condition established by the MONITOR instruction will be cleared.

Effect of MSR IA32_FEATURE_CONTROL MSR

Bits 15:8 of the IA32_FEATURE_CONTROL MSR affect the execution of GETSEC[SENTER]. These bits consist of two fields:

- Bit 15: a global enable control for execution of SENTER.
- Bits 14:8: a parameter control field providing the ability to qualify SENTER execution based on the level of functionality specified with corresponding EDX parameter bits 6:0.

The layout of these fields in the IA32_FEATURE_CONTROL MSR is shown in Table 7-1.

Prior to the execution of GETSEC[SENTER], the lock bit of IA32_FEATURE_CONTROL MSR must be bit set to affirm the settings to be used. Once the lock bit is set, only a power-up reset condition will clear this MSR. The IA32_FEATURE_CONTROL MSR must be configured in accordance to the intended usage at platform initialization. Note that this MSR is only available on SMX or VMX enabled processors. Otherwise, IA32_FEATURE_CONTROL is treated as reserved.

The Intel® Trusted Execution Technology Measured Launched Environment Programming Guide provides additional details and requirements for programming measured environment software to launch in an Intel TXT platform.

Operation in a Uni-Processor Platform

(* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary *)

GETSEC[SENTER] (ILP Only):

```
IF (CR4.SMXE=0)
    THEN #UD;
ELSE IF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSE IF (GETSEC leaf unsupported)
    THEN #UD;
ELSE IF ((in VMX root operation) or
    (CR0.PE=0) or (CR0.CD=1) or (CR0.NW=1) or (CR0.NE=0) or
    (CPL>0) or (EFLAGS.VM=1) or
    (IA32_APIC_BASE.BSP=0) or (TXT chipset not present) or
    (SENTERFLAG=1) or (ACMODEFLAG=1) or (IN_SMM=1) or
    (TPM interface is not present) or
    (EDX ≠ (SENTER_EDX_support_mask & EDX)) or
    (IA32_FEATURE_CONTROL[0]=0) or (IA32_FEATURE_CONTROL[15]=0) or
    ((IA32_FEATURE_CONTROL[14:8] & EDX[6:0]) ≠ EDX[6:0]))
        THEN #GP(0);
IF (GETSEC[PARAMETERS].Parameter_Type = 5, MCA_Handling (bit 6) = 0)
    FOR I = 0 to IA32_MCG_CAP.COUNT-1 DO
        IF IA32_MC[I]_STATUS = uncorrectable error
            THEN #GP(0);
        FI;
    OD;
FI;
IF (IA32_MCG_STATUS.MCIP=1) or (IERR pin is asserted)
```

```
    THEN #GP(0);
ACBASE := EBX;
ACSIZE := ECX;
IF (((ACBASE MOD 4096) ≠ 0) or ((ACSIZE MOD 64) ≠ 0 ) or (ACSIZE < minimum
    module size) or (ACSIZE > AC RAM capacity) or ((ACBASE+ACSIZE) > (2^32 -1)))
        THEN #GP(0);
Mask SMI, INIT, A20M, and NMI external pin events;
SignalTXTMsg(SENTER);
DO
WHILE (no SignalSENTER message);
```

TXT_SENTER__MSG_EVENT (ILP & RLP):

```
Mask and clear SignalSENTER event;
Unmask SignalSEXIT event;
IF (in VMX operation)
    THEN TXT-SHUTDOWN(#IllegalEvent);
FOR I = 0 to IA32_MCG_CAP.COUNT-1 DO
    IF IA32_MC[I]_STATUS = uncorrectable error
        THEN TXT-SHUTDOWN(#UnrecovMCError);
    FI;
OD;
IF (IA32_MCG_STATUS.MCIP=1) or (IERR pin is asserted)
    THEN TXT-SHUTDOWN(#UnrecovMCError);
IF (Voltage or bus ratio status are NOT at a known good state)
    THEN IF (Voltage select and bus ratio are internally adjustable)
        THEN
            Make product-specific adjustment on operating parameters;
        ELSE
            TXT-SHUTDOWN(#IllegalVIDBRatio);
FI;

IA32_MISC_ENABLE := (IA32_MISC_ENABLE & MASK_CONST*)
(* The hexadecimal value of MASK_CONST may vary due to processor implementations *)
A20M := 0;
IA32_DEBUGCTL := 0;
Invalidate processor TLB(s);
Drain outgoing transactions;
Clear performance monitor counters and control;
SENTERFLAG := 1;
SignalTXTMsg(SENTERAck);
IF (logical processor is not ILP)
    THEN GOTO RLP_SENTER_ROUTINE;
(* ILP waits for all logical processors to ACK *)
DO
    DONE := TXT.READ(LT.STS);
WHILE (not DONE);
SignalTXTMsg(SENTERContinue);
SignalTXTMsg(ProcessorHold);
FOR I=ACBASE to ACBASE+ACSIZE-1 DO
    ACRAM[I-ACBASE].ADDR := I;
    ACRAM[I-ACBASE].DATA := LOAD(I);
OD;
IF (ACRAM memory type ≠ WB)
    THEN TXT-SHUTDOWN(#BadACMMType);
```

```
IF (AC module header version is not supported) OR (ACRAM[ModuleType] ≠ 2)
    THEN TXT-SHUTDOWN(#UnsupportedACM);
KEY := GETKEY(ACRAM, ACBASE);
KEYHASH := HASH(KEY);
CSKEYHASH := LT.READ(LT.PUBLIC.KEY);
IF (KEYHASH ≠ CSKEYHASH)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
SIGNATURE := DECRYPT(ACRAM, ACBASE, KEY);
(* The value of SIGNATURE_LEN_CONST is implementation-specific*)
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.I] := SIGNATURE[I];
COMPUTEDSIGNATURE := HASH(ACRAM, ACBASE, ACSIZE);
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.SIGNATURE_LEN_CONST+I] := COMPUTEDSIGNATURE[I];
IF (SIGNATURE ≠ COMPUTEDSIGNATURE)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
ACMCONTROL := ACRAM[CodeControl];
IF ((ACMCONTROL.0 = 0) and (ACMCONTROL.1 = 1) and (snoop hit to modified line detected on ACRAM load))
    THEN TXT-SHUTDOWN(#UnexpectedHITM);
IF (ACMCONTROL reserved bits are set)
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[GDTBasePtr] < (ACRAM[HeaderLen] * 4 + Scratch_size)) OR
    ((ACRAM[GDTBasePtr] + ACRAM[GDTLimit]) >= ACSIZE))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACMCONTROL.0 = 1) and (ACMCONTROL.1 = 1) and (snoop hit to modified
    line detected on ACRAM load))
    THEN ACEntryPoint := ACBASE+ACRAM[ErrorEntryPoint];
ELSE
    ACEntryPoint := ACBASE+ACRAM[EntryPoint];
IF ((ACEntryPoint >= ACSIZE) or (ACEntryPoint < (ACRAM[HeaderLen] * 4 + Scratch_size)))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel] > (ACRAM[GDTLimit] - 15)) or (ACRAM[SegSel] < 8))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel].TI=1) or (ACRAM[SegSel].RPL≠0))
    THEN TXT-SHUTDOWN(#BadACMFormat);

IF (FTM_INTERFACE_ID.[3:0] = 1 ) (* Alternate FTM Interface has been enabled *)
    THEN (* TPM_LOC_CTRL_4 is located at 0FED44008H, TMP_DATA_BUFFER_4 is located at 0FED44080H *)
        WRITE(TPM_LOC_CTRL_4) := 01H; (* Modified HASH.START protocol *)
        (* Write to firmware storage *)
        WRITE(TPM_DATA_BUFFER_4) := SIGNATURE_LEN_CONST + 4;
        FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
            WRITE(TPM_DATA_BUFFER_4 + 2 + I ) := ACRAM[SCRATCH.I];
        WRITE(TPM_DATA_BUFFER_4 + 2 + SIGNATURE_LEN_CONST) := EDX;
        WRITE(FTM.LOC_CTRL) := 06H; (* Modified protocol combining HASH.DATA and HASH.END *)
    ELSE IF (FTM_INTERFACE_ID.[3:0] = 0 ) (* Use standard TPM Interface *)
        ACRAM[SCRATCH.SIGNATURE_LEN_CONST] := EDX;
        WRITE(TPM.HASH.START) := 0;
        FOR I=0 to SIGNATURE_LEN_CONST + 3 DO
            WRITE(TPM.HASH.DATA) := ACRAM[SCRATCH.I];
        WRITE(TPM.HASH.END) := 0;
FI;
ACMODEFLAG := 1;
CR0.[PG.AM.WP] := 0;
```

```
CR4 := 00004000h;
EFLAGS := 00000002h;
IA32_EFER := 0;
EBP := ACBASE;
GDTR.BASE := ACBASE+ACRAM[GDTBasePtr];
GDTR.LIMIT := ACRAM[GDTLimit];
CS.SEL := ACRAM[SegSel];
CS.BASE := 0;
CS.LIMIT := FFFFFh;
CS.G := 1;
CS.D := 1;
CS.AR := 9Bh;
DS.SEL := ACRAM[SegSel]+8;
DS.BASE := 0;
DS.LIMIT := FFFFFh;
DS.G := 1;
DS.D := 1;
DS.AR := 93h;
SS := DS;
ES := DS;
DR7 := 00000400h;
IA32_DEBUGCTL := 0;
SignalTXTMsg(UnlockSMRAM);
SignalTXTMsg(OpenPrivate);
SignalTXTMsg(OpenLocality3);
EIP := ACEntryPoint;
END;
```

RLP_SENTER_ROUTINE: (RLP Only)

```
Mask SMI, INIT, A20M, and NMI external pin events
Unmask SignalWAKEUP event;
Wait for SignalSENTERContinue message;
IA32_APIC_BASE.BSP := 0;
GOTO SENTER sleep state;
END;
```

Flags Affected

All flags are cleared.

Use of Prefixes

LOCK	Causes #UD.
REP*	Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
Operand size	Causes #UD.
NP	66/F2/F3 prefixes are not allowed.
Segment overrides	Ignored.
Address size	Ignored.
REX	Ignored.

Protected Mode Exceptions

#UD	If CR4.SMXE = 0.
	If GETSEC[SENTER] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	If CR0.CD = 1 or CR0.NW = 1 or CR0.NE = 0 or CR0.PE = 0 or CPL > 0 or EFLAGS.VM = 1.
	If in VMX root operation.
	If the initiating processor is not designated as the bootstrap processor via the MSR bit IA32_APIC_BASE.BSP.
	If an Intel® TXT-capable chipset is not present.
	If an Intel® TXT-capable chipset interface to TPM is not detected as present.
	If a protected partition is already active or the processor is already in authenticated code mode.
	If the processor is in SMM.
	If a valid uncorrectable machine check error is logged in IA32_MC[I]_STATUS.
	If the authenticated code base is not on a 4096 byte boundary.
	If the authenticated code size > processor's authenticated code execution area storage capacity.
	If the authenticated code size is not modulo 64.

Real-Address Mode Exceptions

#UD	If CR4.SMXE = 0.
	If GETSEC[SENTER] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	GETSEC[SENTER] is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	If CR4.SMXE = 0.
	If GETSEC[SENTER] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	GETSEC[SENTER] is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

All protected mode exceptions apply.

#GP	If AC code module does not reside in physical address below 2^32 -1.

64-Bit Mode Exceptions

All protected mode exceptions apply.

#GP	If AC code module does not reside in physical address below 2^32 -1.

VM-Exit Condition

Reason (GETSEC)	If in VMX non-root operation.

GETSEC[SEXIT]—Exit Measured Environment

Opcode	Instruction	Description
NP 0F 37 (EAX=5)	GETSEC[SEXIT]	Exit measured environment.

Description

The GETSEC[SEXIT] instruction initiates an exit of a measured environment established by GETSEC[SENTER]. The SEXIT leaf of GETSEC is selected with EAX set to 5 at execution. This instruction leaf sends a message to all logical processors in the platform to signal the measured environment exit.

There are restrictions enforced by the processor for the execution of the GETSEC[SEXIT] instruction:

- Execution is not allowed unless the processor is in protected mode (CR0.PE = 1) with CPL = 0 and EFLAGS.VM = 0.
- The processor must be in a measured environment as launched by a previous GETSEC[SENTER] instruction, but not still in authenticated code execution mode.
- To avoid potential interoperability conflicts between modes, the processor is not allowed to execute this instruction if it currently is in SMM or in VMX operation.
- To ensure consistent handling of SIPI messages, the processor executing the GETSEC[SEXIT] instruction must also be designated the BSP (bootstrap processor) as defined by the register bit IA32_APIC_BASE.BSP (bit 8).

Failure to abide by the above conditions results in the processor signaling a general protection violation.

This instruction initiates a sequence to rendezvous the RLPs with the ILP. It then clears the internal processor flag indicating the processor is operating in a measured environment.

In response to a message signaling the completion of rendezvous, all RLPs restart execution with the instruction that was to be executed at the time GETSEC[SEXIT] was recognized. This applies to all processor conditions, with the following exceptions:

- If an RLP executed HLT and was in this halt state at the time of the message initiated by GETSEC[SEXIT], then execution resumes in the halt state.
- If an RLP was executing MWAIT, then a message initiated by GETSEC[SEXIT] causes an exit of the MWAIT state, falling through to the next instruction.
- If an RLP was executing an intermediate iteration of a string instruction, then the processor resumes execution of the string instruction at the point which the message initiated by GETSEC[SEXIT] was recognized.
- If an RLP is still in the SENTER sleep state (never awakened with GETSEC[WAKEUP]), it will be sent to the wait-for-SIPI state after first clearing the bootstrap processor indicator flag (IA32_APIC_BASE.BSP) and any pending SIPI state. In this case, such RLPs are initialized to an architectural state consistent with having taken a soft reset using the INIT# pin.

Prior to completion of the GETSEC[SEXIT] operation, both the ILP and any active RLPs unmask the response of the external event signals INIT#, A20M, NMI#, and SMI#. This unmasking is performed unconditionally to recognize pin events which are masked after a GETSEC[SENTER]. The state of A20M is unmasked, as the A20M pin is not recognized while the measured environment is active.

On a successful exit of the measured environment, the ILP re-locks the Intel® TXT-capable chipset private configuration space. GETSEC[SEXIT] does not affect the content of any PCR.

At completion of GETSEC[SEXIT] by the ILP, execution proceeds to the next instruction. Since EFLAGS and the debug register state are not modified by this instruction, a pending trap condition is free to be signaled if previously enabled.

Operation in a Uni-Processor Platform

(* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary *)

GETSEC[SEXIT] (ILP Only):

```
IF (CR4.SMXE=0)
    THEN #UD;
ELSE IF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSE IF (GETSEC leaf unsupported)
    THEN #UD;
ELSE IF ((in VMX root operation) or
    (CR0.PE=0) or (CPL>0) or (EFLAGS.VM=1) or
    (IA32_APIC_BASE.BSP=0) or
    (TXT chipset not present) or
    (SENTERFLAG=0) or (ACMODEFLAG=1) or (IN_SMM=1))
        THEN #GP(0);
SignalTXTMsg(SEXIT);
DO
WHILE (no SignalSEXIT message);
```

TXT_SEXIT_MSG_EVENT (ILP & RLP):

```
Mask and clear SignalSEXIT event;
Clear MONITOR FSM;
Unmask SignalSENTER event;
IF (in VMX operation)
    THEN TXT-SHUTDOWN(#IllegalEvent);
SignalTXTMsg(SEXITAck);
IF (logical processor is not ILP)
    THEN GOTO RLP_SEXIT_ROUTINE;
(* ILP waits for all logical processors to ACK *)
DO
    DONE := READ(LT.STS);
WHILE (NOT DONE);
SignalTXTMsg(SEXITContinue);
SignalTXTMsg(ClosePrivate);
SENTERFLAG := 0;
Unmask SMI, INIT, A20M, and NMI external pin events;
END;
```

RLP_SEXIT_ROUTINE (RLPs Only):

```
Wait for SignalSEXITContinue message;
Unmask SMI, INIT, A20M, and NMI external pin events;
IF (prior execution state = HLT)
    THEN reenter HLT state;
IF (prior execution state = SENTER sleep)
    THEN
        IA32_APIC_BASE.BSP := 0;
        Clear pending SIPI state;
        Call INIT_PROCESSOR_STATE;
        Unmask SIPI event;
        GOTO WAIT-FOR-SIPI;
FI;
END;
```

Flags Affected

ILP: None.

RLPs: All flags are modified for an RLP. returning to wait-for-SIPI state, none otherwise.

Use of Prefixes

LOCK	Causes #UD.
REP*	Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
Operand size	Causes #UD.
NP	66/F2/F3 prefixes are not allowed.
Segment overrides	Ignored.
Address size	Ignored.
REX	Ignored.

Protected Mode Exceptions

#UD	If CR4.SMXE = 0.
	If GETSEC[SEXIT] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	If CR0.PE = 0 or CPL > 0 or EFLAGS.VM = 1.
	If in VMX root operation.
	If the initiating processor is not designated via the MSR bit IA32_APIC_BASE.BSP.
	If an Intel® TXT-capable chipset is not present.
	If a protected partition is not already active or the processor is already in authenticated code mode.
	If the processor is in SMM.

Real-Address Mode Exceptions

#UD	If CR4.SMXE = 0.
	If GETSEC[SEXIT] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	GETSEC[SEXIT] is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	If CR4.SMXE = 0.
	If GETSEC[SEXIT] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	GETSEC[SEXIT] is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

All protected mode exceptions apply.

64-Bit Mode Exceptions

All protected mode exceptions apply.

VM-Exit Condition

Reason (GETSEC)	If in VMX non-root operation.

GETSEC[PARAMETERS]—Report the SMX Parameters

Opcode	Instruction	Description
NP 0F 37 (EAX=6)	GETSEC[PARAMETERS]	Report the SMX parameters. The parameters index is input in EBX with the result returned in EAX, EBX, and ECX.

Description

The GETSEC[PARAMETERS] instruction returns specific parameter information for SMX features supported by the processor. Parameter information is returned in EAX, EBX, and ECX, with the input parameter selected using EBX.

Software retrieves parameter information by searching with an input index for EBX starting at 0, and then reading the returned results in EAX, EBX, and ECX. EAX[4:0] is designated to return a parameter type field indicating if a parameter is available and what type it is. If EAX[4:0] is returned with 0, this designates a null parameter and indicates no more parameters are available.

Table 7-7 defines the parameter types supported in current and future implementations.

Table 7-7. SMX Reporting Parameters Format

Parameter Type EAX[4:0]	Parameter Description	EAX[31:5]	EBX[31:0]	ECX[31:0]
0	NULL	Reserved (0 returned)	Reserved (unmodified)	Reserved (unmodified)
1	Supported AC module versions	Reserved (0 returned)	Version comparison mask	Version numbers supported
2	Max size of authenticated code execution area	Multiply by 32 for size in bytes	Reserved (unmodified)	Reserved (unmodified)
3	External memory types supported during AC mode	Memory type bit mask	Reserved (unmodified)	Reserved (unmodified)
4	Selective SENTER functionality control	EAX[14:8] correspond to available SENTER function disable controls	Reserved (unmodified)	Reserved (unmodified)
5	TXT extensions support	TXT Feature Extensions Flags (see Table)	Reserved	Reserved
6-31	Undefined	Reserved (unmodified)	Reserved (unmodified)	Reserved (unmodified)

Table 7-8. TXT Feature Extensions Flags

Bit	Definition	Description
5	Processor based S-CRTM support	Returns 1 if this processor implements a processor-rooted S-CRTM capability and 0 if not (S-CRTM is rooted in BIOS). This flag cannot be used to infer whether the chipset supports TXT or whether the processor support SMX.
6	Machine Check Handling	Returns 1 if it machine check status registers can be preserved through ENTERACCS and SENTER. If this bit is 1, the caller of ENTERACCS and SENTER is not required to clear machine check error status bits before invoking these GETSEC leaves. If this bit returns 0, the caller of ENTERACCS and SENTER must clear all machine check error status bits before invoking these GETSEC leaves.
31:7	Reserved	Reserved for future use. Will return 0.

Supported AC module versions (as defined by the AC module HeaderVersion field) can be determined for a particular SMX capable processor by the type 1 parameter. Using EBX to index through the available parameters reported by GETSEC[PARAMETERS] for each unique parameter set returned for type 1, software can determine the complete list of AC module version(s) supported.

For each parameter set, EBX returns the comparison mask and ECX returns the available HeaderVersion field values supported, after AND'ing the target HeaderVersion with the comparison mask. Software can then determine if a particular AC module version is supported by following the pseudo-code search routine given below:

```
parameter_search_index= 0
do {
        EBX= parameter_search_index++
        EAX= 6
        GETSEC
        if (EAX[4:0] = 1) {
            if ((version_query & EBX) = ECX) {
                version_is_supported= 1
                break
        }
    }
} while (EAX[4:0] ≠ 0)
```

If only AC modules with a HeaderVersion of 0 are supported by the processor, then only one parameter set of type 1 will be returned, as follows: EAX = 00000001H,

EBX = FFFFFFFFH and ECX = 00000000H.

The maximum capacity for an authenticated code execution area supported by the processor is reported with the parameter type of 2. The maximum supported size in bytes is determined by multiplying the returned size in EAX[31:5] by 32. Thus, for a maximum supported authenticated RAM size of 32KBytes, EAX returns with 00008002H.

Supportable memory types for memory mapped outside of the authenticated code execution area are reported with the parameter type of 3. While is active, as initiated by the GETSEC functions SENTER and ENTERACCS and terminated by EXITAC, there are restrictions on what memory types are allowed for the rest of system memory. It is the responsibility of the system software to initialize the memory type range register (MTRR) MSRs and/or the page attribute table (PAT) to only map memory types consistent with the reporting of this parameter. The reporting of supportable memory types of external memory is indicated using a bit map returned in EAX[31:8]. These bit positions correspond to the memory type encodings defined for the MTRR MSR and PAT programming. See Table 7-9.

The parameter type of 4 is used for enumerating the availability of selective GETSEC[SENTER] function disable controls. If a 1 is reported in bits 14:8 of the returned parameter EAX, then this indicates a disable control capability exists with SENTER for a particular function. The enumerated field in bits 14:8 corresponds to use of the EDX input parameter bits 6:0 for SENTER. If an enumerated field bit is set to 1, then the corresponding EDX input parameter bit of EDX may be set to 1 to disable that designated function. If the enumerated field bit is 0 or this parameter is not reported, then no disable capability exists with the corresponding EDX input parameter for SENTER, and EDX bit(s) must be cleared to 0 to enable execution of SENTER. If no selective disable capability for SENTER exists as enumerated, then the corresponding bits in the IA32_FEATURE_CONTROL MSR bits 14:8 must also be programmed to 1 if the SENTER global enable bit 15 of the MSR is set. This is required to enable future extensibility of SENTER selective disable capability with respect to potentially separate software initialization of the MSR.

Table 7-9. External Memory Types Using Parameter 3

EAX Bit Position	Parameter Description
8	Uncacheable (UC)
9	Write Combining (WC)
11:10	Reserved
12	Write-through (WT)

Table 7-9. External Memory Types Using Parameter 3 (Contd.)

13	Write-protected (WP)
14	Write-back (WB)
31:15	Reserved

If the GETSEC[PARAMETERS] leaf or specific parameter is not present for a given SMX capable processor, then default parameter values should be assumed. These are defined in Table 7-10.

Table 7-10. Default Parameter Values

Parameter Type EAX[4:0]	Default Setting	Parameter Description
1	0.0 only	Supported AC module versions.
2	32 KBytes	Authenticated code execution area size.
3	UC only	External memory types supported during AC execution mode.
4	None	Available SENTER selective disable controls.

Operation

```
(* example of a processor supporting only a 0.0 HeaderVersion, 32K ACRAM size, memory types UC and WC *)
IF (CR4.SMXE=0)
    THEN #UD;
ELSE IF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSE IF (GETSEC leaf unsupported)
    THEN #UD;
    (* example of a processor supporting a 0.0 HeaderVersion *)
IF (EBX=0) THEN
    EAX := 00000001h;
    EBX := FFFFFFFFh;
    ECX := 00000000h;
ELSE IF (EBX=1)
    (* example of a processor supporting a 32K ACRAM size *)
    THEN EAX := 00008002h;
ESE IF (EBX= 2)
    (* example of a processor supporting external memory types of UC and WC *)
    THEN EAX := 00000303h;
ESE IF (EBX= other value(s) less than unsupported index value)
    (* EAX value varies. Consult Table 7-7 and Table *)
ELSE (* unsupported index*)
    EAX := 00000000h;
END;
```

Flags Affected

None.

Use of Prefixes

LOCK	Causes #UD.
REP*	Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
Operand size	Causes #UD.

NP	66/F2/F3 prefixes are not allowed.
Segment overrides	Ignored.
Address size	Ignored.
REX	Ignored.

Protected Mode Exceptions

#UD	If CR4.SMXE = 0. If GETSEC[PARAMETERS] is not reported as supported by GETSEC[CAPABILITIES].

Real-Address Mode Exceptions

#UD	If CR4.SMXE = 0. If GETSEC[PARAMETERS] is not reported as supported by GETSEC[CAPABILITIES].

Virtual-8086 Mode Exceptions

#UD	If CR4.SMXE = 0. If GETSEC[PARAMETERS] is not reported as supported by GETSEC[CAPABILITIES].

Compatibility Mode Exceptions

All protected mode exceptions apply.

64-Bit Mode Exceptions

All protected mode exceptions apply.

VM-Exit Condition

Reason (GETSEC)	If in VMX non-root operation.

GETSEC[SMCTRL]—SMX Mode Control

Opcode	Instruction	Description
NP 0F 37 (EAX = 7)	GETSEC[SMCTRL]	Perform specified SMX mode control as selected with the input EBX.

Description

The GETSEC[SMCTRL] instruction is available for performing certain SMX specific mode control operations. The operation to be performed is selected through the input register EBX. Currently only an input value in EBX of 0 is supported. All other EBX settings will result in the signaling of a general protection violation.

If EBX is set to 0, then the SMCTRL leaf is used to re-enable SMI events. SMI is masked by the ILP executing the GETSEC[SENTER] instruction (SMI is also masked in the responding logical processors in response to SENTER rendezvous messages.). The determination of when this instruction is allowed and the events that are unmasked is dependent on the processor context (See Table 7-11). For brevity, the usage of SMCTRL where EBX=0 will be referred to as GETSEC[SMCTRL(0)].

As part of support for launching a measured environment, the SMI, NMI, and INIT events are masked after GETSEC[SENTER], and remain masked after exiting authenticated execution mode. Unmasking these events should be accompanied by securely enabling these event handlers. These security concerns can be addressed in VMX operation by a MVMM.

The VM monitor can choose two approaches:

- In a dual monitor approach, the executive software will set up an SMM monitor in parallel to the executive VMM (i.e., the MVMM), see Chapter 34, "System Management Mode," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C. The SMM monitor is dedicated to handling SMI events without compromising the security of the MVMM. This usage model of handling SMI while a measured environment is active does not require the use of GETSEC[SMCTRL(0)] as event re-enabling after the VMX environment launch is handled implicitly and through separate VMX based controls.
- If a dedicated SMM monitor will not be established and SMIs are to be handled within the measured environment, then GETSEC[SMCTRL(0)] can be used by the executive software to re-enable SMI that has been masked as a result of SENTER.

Table 7-11 defines the processor context in which GETSEC[SMCTRL(0)] can be used and which events will be unmasked. Note that the events that are unmasked are dependent upon the currently operating processor context.

Table 7-11. Supported Actions for GETSEC[SMCTRL(0)]

ILP Mode of Operation	SMCTRL execution action
In VMX non-root operation	VM exit
SENTERFLAG = 0	#GP(0), illegal context
In authenticated code execution mode (ACMODEFLAG = 1)	#GP(0), illegal context
SENTERFLAG = 1, not in VMX operation, not in SMM	Unmask SMI
SENTERFLAG = 1, in VMX root operation, not in SMM	Unmask SMI if SMM monitor is not configured, otherwise #GP(0)
SENTERFLAG = 1, In VMX root operation, in SMM	#GP(0), illegal context

Operation

```
(* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary *)
IF (CR4.SMXE=0)
    THEN #UD;
ELSE IF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSE IF (GETSEC leaf unsupported)
    THEN #UD;
ELSE IF ((CR0.PE=0) or (CPL>0) OR (EFLAGS.VM=1))
    THEN #GP(0);
ELSE IF((EBX=0) and (SENTERFLAG=1) and (ACMODEFLAG=0) and (IN_SMM=0) and
        (((in VMX root operation) and (SMM monitor not configured)) or (not in VMX operation)) )
    THEN unmask SMI;
ELSE
    #GP(0);
END
```

Flags Affected

None.

Use of Prefixes

LOCK	Causes #UD.
REP*	Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
Operand size	Causes #UD.
NP	66/F2/F3 prefixes are not allowed.
Segment overrides	Ignored.
Address size	Ignored.
REX	Ignored.

Protected Mode Exceptions

#UD	If CR4.SMXE = 0.
	If GETSEC[SMCTRL] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	If CR0.PE = 0 or CPL > 0 or EFLAGS.VM = 1.
	If in VMX root operation.
	If a protected partition is not already active or the processor is currently in authenticated code mode.
	If the processor is in SMM.
	If the SMM monitor is not configured.

Real-Address Mode Exceptions

#UD	If CR4.SMXE = 0.
	If GETSEC[SMCTRL] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	GETSEC[SMCTRL] is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	If CR4.SMXE = 0.
	If GETSEC[SMCTRL] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	GETSEC[SMCTRL] is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

All protected mode exceptions apply.

64-Bit Mode Exceptions

All protected mode exceptions apply.

VM-exit Condition

Reason (GETSEC) If in VMX non-root operation.

GETSEC[WAKEUP]—Wake Up Sleeping Processors in Measured Environment

Opcode	Instruction	Description
NP 0F 37 (EAX=8)	GETSEC[WAKEUP]	Wake up the responding logical processors from the SENTER sleep state.

Description

The GETSEC[WAKEUP] leaf function broadcasts a wake-up message to all logical processors currently in the SENTER sleep state. This GETSEC leaf must be executed only by the ILP, in order to wake-up the RLPs. Responding logical processors (RLPs) enter the SENTER sleep state after completion of the SENTER rendezvous sequence.

The GETSEC[WAKEUP] instruction may only be executed:

- In a measured environment as initiated by execution of GETSEC[SENTER].
- Outside of authenticated code execution mode.
- Execution is not allowed unless the processor is in protected mode with CPL = 0 and EFLAGS.VM = 0.
- In addition, the logical processor must be designated as the boot-strap processor as configured by setting IA32_APIC_BASE.BSP = 1.

If these conditions are not met, attempts to execute GETSEC[WAKEUP] result in a general protection violation.

An RLP exits the SENTER sleep state and start execution in response to a WAKEUP signal initiated by ILP's execution of GETSEC[WAKEUP]. The RLP retrieves a pointer to a data structure that contains information to enable execution from a defined entry point. This data structure is located using a physical address held in the Intel® TXT-capable chipset configuration register LT.MLE.JOIN. The register is publicly writable in the chipset by all processors and is not restricted by the Intel® TXT-capable chipset configuration register lock status. The format of this data structure is defined in Table 7-12.

Table 7-12. RLP MVMM JOIN Data Structure

Offset	Field
0	GDT limit
4	GDT base pointer
8	Segment selector initializer
12	EIP

The MLE JOIN data structure contains the information necessary to initialize RLP processor state and permit the processor to join the measured environment. The GDTR, LIP, and CS, DS, SS, and ES selector values are initialized using this data structure. The CS selector index is derived directly from the segment selector initializer field; DS, SS, and ES selectors are initialized to CS+8. The segment descriptor fields are initialized implicitly with BASE = 0, LIMIT = FFFFFH, G = 1, D = 1, P = 1, S = 1; read/write/access for DS, SS, and ES; and execute/read/access for CS. It is the responsibility of external software to establish a GDT pointed to by the MLE JOIN data structure that contains descriptor entries consistent with the implicit settings initialized by the processor (see Table 7-6). Certain states from the content of Table 7-12 are checked for consistency by the processor prior to execution. A failure of any consistency check results in the RLP aborting entry into the protected environment and signaling an Intel® TXT shutdown condition. The specific checks performed are documented later in this section. After successful completion of processor consistency checks and subsequent initialization, RLP execution in the measured environment begins from the entry point at offset 12 (as indicated in Table 7-12).

Operation

```
(* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary *)
IF (CR4.SMXE=0)
    THEN #UD;
ELSE IF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSE IF (GETSEC leaf unsupported)
    THEN #UD;
ELSE IF ((CR0.PE=0) or (CPL>0) or (EFLAGS.VM=1) or (SENTERFLAG=0) or (ACMODEFLAG=1) or (IN_SMM=0) or (in VMX operation) or
(IA32_APIC_BASE.BSP=0) or (TXT chipset not present))
    THEN #GP(0);
ELSE
    SignalTXTMsg(WAKEUP);
END;
```

RLP_SIPI_WAKEUP_FROM_SENTER_ROUTINE: (RLP Only)

```
WHILE (no SignalWAKEUP event);
IF (IA32_SMM_MONITOR_CTL[0] ≠ ILP.IA32_SMM_MONITOR_CTL[0])
    THEN TXT-SHUTDOWN(#IllegalEvent)
IF (IA32_SMM_MONITOR_CTL[0] = 0)
    THEN Unmask SMI pin event;
ELSE
    Mask SMI pin event;
Mask A20M, and NMI external pin events (unmask INIT);
Mask SignalWAKEUP event;
Invalidate processor TLB(s);
Drain outgoing transactions;
TempGDTRLIMIT := LOAD(LT.MLE.JOIN);
TempGDTRBASE := LOAD(LT.MLE.JOIN+4);
TempSegSel := LOAD(LT.MLE.JOIN+8);
TempEIP := LOAD(LT.MLE.JOIN+12);
IF (TempGDTLimit & FFFF0000h)
    THEN TXT-SHUTDOWN(#BadJOINFormat);
IF ((TempSegSel > TempGDTRLIMIT-15) or (TempSegSel < 8))
    THEN TXT-SHUTDOWN(#BadJOINFormat);
IF ((TempSegSel.TI=1) or (TempSegSel.RPL≠0))
    THEN TXT-SHUTDOWN(#BadJOINFormat);
CR0.[PG,CD,NW,AM,WP] := 0;
CR0.[NE,PE] := 1;
CR4 := 00004000h;
EFLAGS := 00000002h;
IA32_EFER := 0;
GDTR.BASE := TempGDTRBASE;
GDTR.LIMIT := TempGDTRLIMIT;
CS.SEL := TempSegSel;
CS.BASE := 0;
CS.LIMIT := FFFFFh;
CS.G := 1;
CS.D := 1;
CS.AR := 9Bh;
DS.SEL := TempSegSel+8;
DS.BASE := 0;
DS.LIMIT := FFFFFh;
DS.G := 1;
```

```
DS.D := 1;
DS.AR := 93h;
SS := DS;
ES := DS;
DR7 := 00000400h;
IA32_DEBUGCTL := 0;
EIP := TempEIP;
END;
```

Flags Affected

None.

Use of Prefixes

LOCK	Causes #UD.
REP*	Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
Operand size	Causes #UD.
NP	66/F2/F3 prefixes are not allowed.
Segment overrides	Ignored.
Address size	Ignored.
REX	Ignored.

Protected Mode Exceptions

#UD	If CR4.SMXE = 0.
	If GETSEC[WAKEUP] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	If CR0.PE = 0 or CPL > 0 or EFLAGS.VM = 1.
	If in VMX operation.
	If a protected partition is not already active or the processor is currently in authenticated code mode.
	If the processor is in SMM.
#UD	If CR4.SMXE = 0.
	If GETSEC[WAKEUP] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	GETSEC[WAKEUP] is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	If CR4.SMXE = 0.
	If GETSEC[WAKEUP] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	GETSEC[WAKEUP] is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

All protected mode exceptions apply.

64-Bit Mode Exceptions

All protected mode exceptions apply.

VM-exit Condition

Reason (GETSEC)	If in VMX non-root operation.

CHAPTER 8
INSTRUCTION SET REFERENCE UNIQUE TO INTEL® XEON PHI™ PROCESSORS

This chapter describes the instruction set that is unique to Intel® Xeon Phi™ Processors based on the Knights Landing and Knights Mill microarchitectures. The set is not supported in any other Intel processors. Included are Intel® AVX-512 instructions. For additional instructions supported on these processors, see Chapter 3, "Instruction Set Reference, A-L"; Chapter 4, "Instruction Set Reference, M-U"; Chapter 5, "Instruction Set Reference, V"; and Chapter 6, "Instruction Set Reference, W-Z."

PREFETCHWT1—Prefetch Vector Data Into Caches With Intent to Write and T1 Hint

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 0D /2 PREFETCHWT1 m8	M	V/V	PREFETCHWT1	Move data from m8 closer to the processor using T1 hint with intent to write.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

Description

Fetches the line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy specified by an intent to write hint (so that data is brought into 'Exclusive' state via a request for ownership) and a locality hint:

• T1 (temporal data with respect to first level cache)—prefetch data into the second level cache.

The source operand is a byte memory location. (The locality hints are encoded into the machine level instruction using bits 3 through 5 of the ModR/M byte. Use of any ModR/M value other than the specified ones will lead to unpredictable behavior.)

If the line selected is already present in the cache hierarchy at a level closer to the processor, no data movement occurs. Prefetches from uncacheable or WC memory are ignored.

The PREFETCHWT1 instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor in anticipation of future use.

The implementation of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes. Additional details of the implementation-dependent locality hints are described in Section 9.5, "Memory Optimization Using Prefetch" of the Intel® 64 and IA-32 Architectures Optimization Reference Manual.

It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type that permits speculative reads (that is, the WB, WC, and WT memory types). A PREFETCHWT1 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 PREFETCHWT1 instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCHWT1 instruction is also unordered with respect to CLFLUSH and CLFLUSHOPT instructions, other PREFETCHWT1 instructions, or any other general instruction. It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

```
Prefetch (m8, Level = 1, EXCLUSIVE=1);
```

Flags Affected

All flags are affected.

C/C++ Compiler Intrinsic Equivalent

void _mm_prefetch(char const *, int hint= _MM_HINT_ET1);

Protected Mode Exceptions

#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.

V4FMADDPS/V4FNMADDPS—Packed Single Precision Floating-Point Fused Multiply-Add (4-Iterations)

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.F2.0F38.W0 9A /r V4FMADDPS zmm1{k1}{z}, zmm2+3, m128	A	V/V	AVX512_4FMAPS	Multiply packed single precision floating-point values from source register block indicated by zmm2 by values from m128 and accumulate the result in zmm1.
EVEX.512.F2.0F38.W0 AA /r V4FNMADDPS zmm1{k1}{z}, zmm2+3, m128	A	V/V	AVX512_4FMAPS	Multiply and negate packed single precision floating-point values from source register block indicated by zmm2 by values from m128 and accumulate the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1_4X	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction computes 4 sequential packed fused single precision floating-point multiply-add instructions with a sequentially selected memory operand in each of the four steps.

In the above box, the notation of "+3" is used to denote that the instruction accesses 4 source registers based on that operand; sources are consecutive, start in a multiple of 4 boundary, and contain the encoded register operand.

This instruction supports memory fault suppression. The entire memory operand is loaded if any of the 16 lowest significant mask bits is set to 1 or if a "no masking" encoding is used.

The tuple type Tuple1_4X implies that four 32-bit elements (16 bytes) are referenced by the memory operation portion of this instruction.

Rounding is performed at every FMA (fused multiply and add) boundary. Exceptions are also taken sequentially. Pre- and post-computational exceptions of the first FMA take priority over the pre- and post-computational exceptions of the second FMA, etc.

Operation

src_reg_id is the 5 bit index of the vector register specified in the instruction as the src1 register.

```
define NFMA_PS(kl, vl, dest, k1, msrc, regs_loaded, src_base, posneg):
    tmpdest := dest

    // reg[] is an array representing the SIMD register file.
    FOR j := 0 to regs_loaded-1:
        FOR i := 0 to kl-1:
            IF k1[i] or *no writemask*:
                IF posneg = 0:
                    tmpdest.single[i] := RoundFPControl_MXCSR(tmpdest.single[i] - reg[src_base + j ].single[i] * msrc.single[j])
                ELSE:
                    tmpdest.single[i] := RoundFPControl_MXCSR(tmpdest.single[i] + reg[src_base + j ].single[i] * msrc.single[j])
            ELSE IF *zeroing*:
                tmpdest.single[i] := 0
    dest := tmpdst
    dest[MAX_VL-1:VL] := 0

V4FMADDPS and V4FNMADDPS dest{k1}, src1, msrc (AVX512)
KL, VL = (16,512)

regs_loaded := 4
src_base := src_reg_id & ~3 // for src1 operand
posneg := 0 if negative form, 1 otherwise
NFMA_PS(kl, vl, dest, k1, msrc, regs_loaded, src_base, posneg)
```

Intel C/C++ Compiler Intrinsic Equivalent

V4FMADDPS __m512 _mm512_4fmadd_ps(__m512, __m512x4, __m128 *);
V4FMADDPS __m512 _mm512_mask_4fmadd_ps(__m512, __mmask16, __m512x4, __m128 *);
V4FMADDPS __m512 _mm512_maskz_4fmadd_ps(__mmask16, __m512, __m512x4, __m128 *);
V4FNMADDPS __m512 _mm512_4fnmadd_ps(__m512, __m512x4, __m128 *);
V4FNMADDPS __m512 _mm512_mask_4fnmadd_ps(__m512, __mmask16, __m512x4, __m128 *);
V4FNMADDPS __m512 _mm512_maskz_4fnmadd_ps(__mmask16, __m512, __m512x4, __m128 *);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Type E2; additionally:

#UD	If the EVEX broadcast bit is set to 1.
#UD	If the MODRM.mod = 0b11.

V4FMADDSS/V4FNMADDSS—Scalar Single Precision Floating-Point Fused Multiply-Add (4-Iterations)

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F2.0F38.W0 9B /r V4FMADDSS xmm1{k1}{z}, xmm2+3, m128	A	V/V	AVX512_4FMAPS	Multiply scalar single precision floating-point values from source register block indicated by xmm2 by values from m128 and accumulate the result in xmm1.
EVEX.LLIG.F2.0F38.W0 AB /r V4FNMADDSS xmm1{k1}{z}, xmm2+3, m128	A	V/V	AVX512_4FMAPS	Multiply and negate scalar single precision floating-point values from source register block indicated by xmm2 by values from m128 and accumulate the result in xmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1_4X	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction computes 4 sequential scalar fused single precision floating-point multiply-add instructions with a sequentially selected memory operand in each of the four steps.

In the above box, the notation of "+3" is used to denote that the instruction accesses 4 source registers based that operand; sources are consecutive, start in a multiple of 4 boundary, and contain the encoded register operand.

This instruction supports memory fault suppression. The entire memory operand is loaded if the least significant mask bit is set to 1 or if a "no masking" encoding is used.

The tuple type Tuple1_4X implies that four 32-bit elements (16 bytes) are referenced by the memory operation portion of this instruction.

Rounding is performed at every FMA boundary. Exceptions are also taken sequentially. Pre- and post-computational exceptions of the first FMA take priority over the pre- and post-computational exceptions of the second FMA, etc.

Operation

src_reg_id is the 5 bit index of the vector register specified in the instruction as the src1 register.

```
define NFMA_SS(vl, dest, k1, msrc, regs_loaded, src_base, posneg):
    tmpdest := dest
    // reg[] is an array representing the SIMD register file.
    IF k1[0] or *no writemask*:
        FOR j := 0 to regs_loaded - 1:
            IF posneg = 0:
                tmpdest.single[0] := RoundFPControl_MXCSR(tmpdest.single[0] - reg[src_base + j ].single[0] * msrc.single[j])
            ELSE:
                tmpdest.single[0] := RoundFPControl_MXCSR(tmpdest.single[0] + reg[src_base + j ].single[0] * msrc.single[j])
    ELSE IF *zeroing*:
        tmpdest.single[0] := 0
    dest := tmpdst
    dest[MAX_VL-1:VL] := 0
```

V4FMADDSS and V4FNMADDSS dest{k1}, src1, msrc (AVX512)

```
VL = 128

regs_loaded := 4
src_base := src_reg_id & ~3 // for src1 operand
posneg := 0 if negative form, 1 otherwise
NFMA_SS(vl, dest, k1, msrc, regs_loaded, src_base, posneg)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
V4FMADDSS __m128 _mm_4fmadd_ss(__m128, __m128x4, __m128 *);
V4FMADDSS __m128 _mm_mask_4fmadd_ss(__m128, __mmask8, __m128x4, __m128 *);
V4FMADDSS __m128 _mm_maskz_4fmadd_ss(__mmask8, __m128, __m128x4, __m128 *);
V4FNMADDSS __m128 _mm_4fnmadd_ss(__m128, __m128x4, __m128 *);
V4FNMADDSS __m128 _mm_mask_4fnmadd_ss(__m128, __mmask8, __m128x4, __m128 *);
V4FNMADDSS __m128 _mm_maskz_4fnmadd_ss(__mmask8, __m128, __m128x4, __m128 *);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Type E2; additionally:

#UD	If the EVEX broadcast bit is set to 1.
#UD	If the MODRM.mod = 0b11.

VEXP2PD—Approximation to the Exponential 2^x of Packed Double Precision Floating-Point Values With Less Than 2^-23 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 C8 /r VEXP2PD zmm1 {k1}{z}, zmm2/m512/m64bcst {sae}	A	V/V	AVX512ER	Computes approximations to the exponential 2^x (with less than 2^-23 of maximum relative error) of the packed double precision floating-point values from zmm2/m512/m64bcst and stores the floating-point result in zmm1with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A

Description

Computes the approximate base-2 exponential evaluation of the double precision floating-point values in the source operand (the second operand) and stores the results to the destination operand (the first operand) using the writemask k1. The approximate base-2 exponential is evaluated with less than 2^-23 of relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FTZ.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

A numerically exact implementation of VEXP2xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VEXP2PD

```
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+63:i] := EXP2_23_DP(SRC[63:0])
                ELSE DEST[i+63:i] := EXP2_23_DP(SRC[i+63:i])
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+63:i] := 0
        FI;
    FI;
ENDFOR;
```

Table 8-1. Special Values Behavior

Source Input	Result	Comments
NaN	QNaN(src)	If (SRC = SNaN) then #I
+∞	+∞	
+/-0	1.0f	Exact result
-∞	+0.0f	
Integral value N	2^ (N)	Exact result

Intel C/C++ Compiler Intrinsic Equivalent

VEXP2PD __m512d _mm512_exp2a23_round_pd (__m512d a, int sae);
VEXP2PD __m512d _mm512_mask_exp2a23_round_pd (__m512d a, __mmask8 m, __m512d b, int sae);
VEXP2PD __m512d _mm512_maskz_exp2a23_round_pd (__mmask8 m, __m512d b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Overflow.

Other Exceptions

See Table 2-48, "Type E2 Class Exception Conditions."

VEXP2PS—Approximation to the Exponential 2^x of Packed Single Precision Floating-Point Values With Less Than 2^-23 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C8 /r VEXP2PS zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	A	V/V	AVX512ER	Computes approximations to the exponential 2^x (with less than 2^-23 of maximum relative error) of the packed single precision floating-point values from zmm2/m512/m32bcst and stores the floating-point result in zmm1with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A

Description

Computes the approximate base-2 exponential evaluation of the single precision floating-point values in the source operand (the second operand) and store the results in the destination operand (the first operand) using the write-mask k1. The approximate base-2 exponential is evaluated with less than 2^-23 of relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FTZ.

The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

A numerically exact implementation of VEXP2xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VEXP2PS

```
(KL, VL) = (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+31:i] := EXP2_23_SP(SRC[31:0])
                ELSE DEST[i+31:i] := EXP2_23_SP(SRC[i+31:i])
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+31:i] := 0
        FI;
    FI;
ENDFOR;
```

Table 8-2. Special Values Behavior

Source Input	Result	Comments
NaN	QNaN(src)	If (SRC = SNaN) then #I
+∞	+∞	
+/-0	1.0f	Exact result
-∞	+0.0f	
Integral value N	2^ (N)	Exact result

Intel C/C++ Compiler Intrinsic Equivalent

VEXP2PS __m512 _mm512_exp2a23_round_ps (__m512 a, int sae);
VEXP2PS __m512 _mm512_mask_exp2a23_round_ps (__m512 a, __mmask16 m, __m512 b, int sae);
VEXP2PS __m512 _mm512_maskz_exp2a23_round_ps (__mmask16 m, __m512 b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Overflow.

Other Exceptions

See Table 2-48, "Type E2 Class Exception Conditions."

VGATHERPF0DPS/VGATHERPF0QPS/VGATHERPF0DPD/VGATHERPF0QPD—Sparse Prefetch Packed SP/DP Data Values With Signed Dword, Signed Qword Indices Using T0 Hint

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C6 /1 /vsib VGATHERPF0DPS vm32z {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing single precision data using opmask k1 and T0 hint.
EVEX.512.66.0F38.W0 C7 /1 /vsib VGATHERPF0QPS vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing single precision data using opmask k1 and T0 hint.
EVEX.512.66.0F38.W1 C6 /1 /vsib VGATHERPF0DPD vm32y {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing double precision data using opmask k1 and T0 hint.
EVEX.512.66.0F38.W1 C7 /1 /vsib VGATHERPF0QPD vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing double precision data using opmask k1 and T0 hint.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	N/A	N/A	N/A

Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

Lines prefetched are loaded into to a location in the cache hierarchy specified by a locality hint (T0):

• T0 (temporal data)—prefetch data into the first level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

(1) The prefetches may happen in any order (or not at all). The instruction is a hint.

(2) The mask is left unchanged.

(3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.

(4) No FP nor memory faults may be produced by this instruction.

(5) Prefetches do not handle cache line splits

(6) A #UD is signaled if the memory operand is encoded without the SIB byte.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist.
VINDEX stands for the memory operand vector of indices (a vector register).
SCALE stands for the memory operand scalar (1, 2, 4 or 8).
DISP is the optional 1, 2 or 4 byte displacement.
PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

VGATHERPF0DPS (EVEX Encoded Version)

```
(KL, VL) = (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR
```

VGATHERPF0DPD (EVEX Encoded Version)

```
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR
```

VGATHERPF0QPS (EVEX Encoded Version)

```
(KL, VL) = (8, 256)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR
```

VGATHERPF0QPD (EVEX Encoded Version)

```
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalent

VGATHERPF0DPD void _mm512_mask_prefetch_i32gather_pd(__m256i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF0DPS void _mm512_mask_prefetch_i32gather_ps(__m512i vdx, __mmask16 m, void * base, int scale, int hint);
VGATHERPF0QPD void _mm512_mask_prefetch_i64gather_pd(__m512i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF0QPS void _mm512_mask_prefetch_i64gather_ps(__m512i vdx, __mmask8 m, void * base, int scale, int hint);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-64, "Type E12NP Class Exception Conditions."

VGATHERPF1DPS/VGATHERPF1QPS/VGATHERPF1DPD/VGATHERPF1QPD—Sparse Prefetch Packed SP/DP Data Values With Signed Dword, Signed Qword Indices Using T1 Hint

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C6 /2 /vsib VGATHERPF1DPS vm32z {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing single precision data using opmask k1 and T1 hint.
EVEX.512.66.0F38.W0 C7 /2 /vsib VGATHERPF1QPS vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing single precision data using opmask k1 and T1 hint.
EVEX.512.66.0F38.W1 C6 /2 /vsib VGATHERPF1DPD vm32y {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing double precision data using opmask k1 and T1 hint.
EVEX.512.66.0F38.W1 C7 /2 /vsib VGATHERPF1QPD vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing double precision data using opmask k1 and T1 hint.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	N/A	N/A	N/A

Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

Lines prefetched are loaded into to a location in the cache hierarchy specified by a locality hint (T1):

• T1 (temporal data)—prefetch data into the second level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

(1) The prefetches may happen in any order (or not at all). The instruction is a hint.

(2) The mask is left unchanged.

(3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.

(4) No FP nor memory faults may be produced by this instruction.

(5) Prefetches do not handle cache line splits

(6) A #UD is signaled if the memory operand is encoded without the SIB byte.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist.
VINDEX stands for the memory operand vector of indices (a vector register).
SCALE stands for the memory operand scalar (1, 2, 4 or 8).
DISP is the optional 1, 2 or 4 byte displacement.
PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

VGATHERPF1DPS (EVEX Encoded Version)

```
(KL, VL) = (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=1, RFO = 0)
    FI;
ENDFOR
```

VGATHERPF1DPD (EVEX Encoded Version)

```
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=1, RFO = 0)
    FI;
ENDFOR
```

VGATHERPF1QPS (EVEX Encoded Version)

```
(KL, VL) = (8, 256)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=1, RFO = 0)
    FI;
ENDFOR
```

VGATHERPF1QPD (EVEX Encoded Version)

```
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=1, RFO = 0)
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalent

VGATHERPF1DPD void _mm512_mask_prefetch_i32gather_pd(__m256i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF1DPS void _mm512_mask_prefetch_i32gather_ps(__m512i vdx, __mmask16 m, void * base, int scale, int hint);
VGATHERPF1QPD void _mm512_mask_prefetch_i64gather_pd(__m512i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF1QPS void _mm512_mask_prefetch_i64gather_ps(__m512i vdx, __mmask8 m, void * base, int scale, int hint);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-64, "Type E12NP Class Exception Conditions."

VP4DPWSSDS—Dot Product of Signed Words With Dword Accumulation and Saturation (4-Iterations)

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.F2.0F38.W0 53 /r VP4DPWSSDS zmm1{k1}{z}, zmm2+3, m128	A	V/V	AVX512_4VNNIW	Multiply signed words from source register block indicated by zmm2 by signed words from m128 and accumulate the resulting dword results with signed saturation in zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1_4X	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction computes 4 sequential register source-block dot-products of two signed word operands with doubleword accumulation and signed saturation. The memory operand is sequentially selected in each of the four steps.

In the above box, the notation of "+3" is used to denote that the instruction accesses 4 source registers based on that operand; sources are consecutive, start in a multiple of 4 boundary, and contain the encoded register operand.

This instruction supports memory fault suppression. The entire memory operand is loaded if any bit of the lowest 16-bits of the mask is set to 1 or if a "no masking" encoding is used.

The tuple type Tuple1_4X implies that four 32-bit elements (16 bytes) are referenced by the memory operation portion of this instruction.

Operation

src_reg_id is the 5 bit index of the vector register specified in the instruction as the src1 register.

VP4DPWSSDS dest, src1, src2

```
(KL,VL) = (16,512)
N := 4

ORIGDEST := DEST
src_base := src_reg_id & ~ (N-1) // for src1 operand

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        FOR m := 0 to N-1:
            t := SRC2.dword[m]
            p1dword := reg[src_base+m].word[2*i] * t.word[0]
            p2dword := reg[src_base+m].word[2*i+1] * t.word[1]
            DEST.dword[i] := SIGNED_DWORD_SATURATE(DEST.dword[i] + p1dword + p2dword)
    ELSE IF *zeroing*:
        DEST.dword[i] := 0
    ELSE
        DEST.dword[i] := ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VP4DPWSSDS __m512i _mm512_4dpwssds_epi32(__m512i, __m512ix4, __m128i *);
VP4DPWSSDS __m512i _mm512_mask_4dpwssds_epi32(__m512i, __mmask16, __m512ix4, __m128i *);
VP4DPWSSDS __m512i _mm512_maskz_4dpwssds_epi32(__mmask16, __m512i, __m512ix4, __m128i *);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Type E4; additionally:

#UD	If the EVEX broadcast bit is set to 1.
#UD	If the MODRM.mod = 0b11.

VP4DPWSSD—Dot Product of Signed Words With Dword Accumulation (4-Iterations)

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.F2.0F38.W0 52 /r VP4DPWSSD zmm1{k1}{z}, zmm2+3, m128	A	V/V	AVX512_4VNNIW	Multiply signed words from source register block indicated by zmm2 by signed words from m128 and accumulate resulting signed dwords in zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1_4X	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction computes 4 sequential register source-block dot-products of two signed word operands with doubleword accumulation; see Figure 8-1 below. The memory operand is sequentially selected in each of the four steps.

In the above box, the notation of "+3"' is used to denote that the instruction accesses 4 source registers based on that operand; sources are consecutive, start in a multiple of 4 boundary, and contain the encoded register operand.

This instruction supports memory fault suppression. The entire memory operand is loaded if any bit of the lowest 16-bits of the mask is set to 1 or if a "no masking" encoding is used.

The tuple type Tuple1_4X implies that four 32-bit elements (16 bytes) are referenced by the memory operation portion of this instruction.

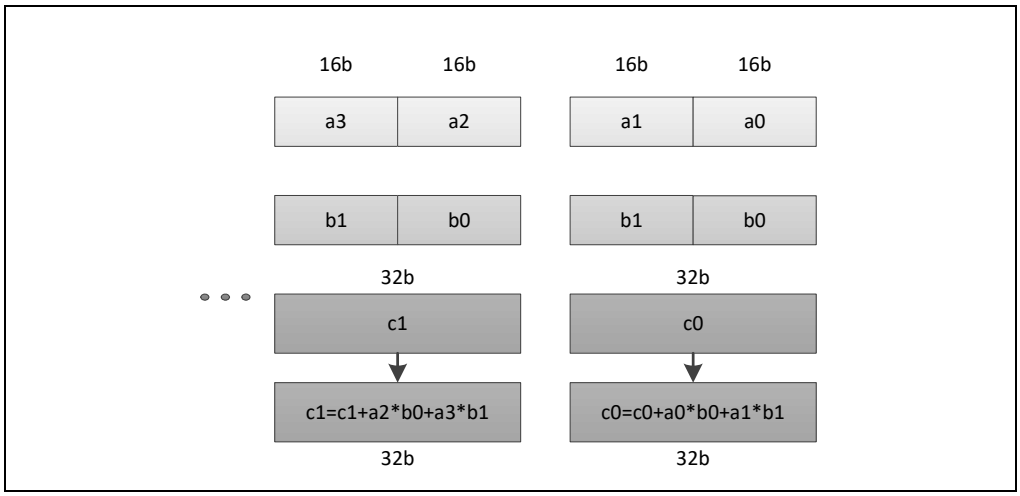


Figure 8-1. Register Source-Block Dot Product of Two Signed Word Operands With Doubleword Accumulation[1]

NOTES:

1. For illustration purposes, one source-block dot product instance is shown out of the four.

Operation

src_reg_id is the 5 bit index of the vector register specified in the instruction as the src1 register.

```
VP4DPWSSD dest, src1, src2
(KL,VL) = (16,512)
N := 4

ORIGDEST := DEST
src_base := src_reg_id & ~ (N-1) // for src1 operand

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        FOR m := 0 to N-1:
            t := SRC2.dword[m]
            p1dword := reg[src_base+m].word[2*i] * t.word[0]
            p2dword := reg[src_base+m].word[2*i+1] * t.word[1]
            DEST.dword[i] := DEST.dword[i] + p1dword + p2dword
    ELSE IF *zeroing*:
        DEST.dword[i] := 0
    ELSE
        DEST.dword[i] := ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VP4DPWSSD __m512i _mm512_4dpwssd_epi32(__m512i, __m512ix4, __m128i *);
VP4DPWSSD __m512i _mm512_mask_4dpwssd_epi32(__m512i, __mmask16, __m512ix4, __m128i *);
VP4DPWSSD __m512i _mm512_maskz_4dpwssd_epi32(__mmask16, __m512i, __m512ix4, __m128i *);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Type E4; additionally:

#UD	If the EVEX broadcast bit is set to 1.
#UD	If the MODRM.mod = 0b11.

VRCP28PD—Approximation to the Reciprocal of Packed Double Precision Floating-Point Values With Less Than 2^-28 Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 CA /r VRCP28PD zmm1 {k1}{z}, zmm2/m512/m64bcst {sae}	A	V/V	AVX512ER	Computes the approximate reciprocals (< 2^-28 relative error) of the packed double precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1. Under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Computes the reciprocal approximation of the float64 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FTZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is ±∞, ±0.0 is returned for that element. Also, if any source element is ±0.0, ±∞ is returned for that element.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

A numerically exact implementation of VRCP28xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP28PD (EVEX Encoded Versions)

```
(KL, VL) = (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+63:i] := RCP_28_DP(1.0/SRC[63:0]);
                ELSE DEST[i+63:i] := RCP_28_DP(1.0/SRC[i+63:i]);
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+63:i] := 0
        FI;
    FI;
ENDFOR;
```

Table 8-3. VRCP28PD Special Cases

Input Value	Result Value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$0 \le X < 2^{-1022}$	INF	Positive input denormal or zero; #Z
$-2^{-1022} < X \le -0$	-INF	Negative input denormal or zero; #Z
$X > 2^{1022}$	+0.0f	
$X < -2^{1022}$	-0.0f	
$X = +\infty$	+0.0f	
$X = -\infty$	-0.0f	
$X = 2^{-n}$	2^{n}	Exact result (unless input/output is a denormal)
$X = -2^{-n}$	-2^{n}	Exact result (unless input/output is a denormal)

Intel C/C++ Compiler Intrinsic Equivalent

VRCP28PD __m512d _mm512_rcp28_round_pd (__m512d a, int sae);
VRCP28PD __m512d _mm512_mask_rcp28_round_pd(__m512d a, __mmask8 m, __m512d b, int sae);
VRCP28PD __m512d _mm512_maskz_rcp28_round_pd(__mmask8 m, __m512d b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero.

Other Exceptions

See Table 2-48, "Type E2 Class Exception Conditions."

VRCP28SD—Approximation to the Reciprocal of Scalar Double Precision Floating-Point Value With Less Than 2^-28 Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W1 CB /r VRCP28SD xmm1 {k1}{z}, xmm2, xmm3/m64 {sae}	A	V/V	AVX512ER	Computes the approximate reciprocal (< 2^-28 relative error) of the scalar double precision floating-point value in xmm3/m64 and stores the results in xmm1. Under writemask. Also, upper double precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Computes the reciprocal approximation of the low float64 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error. The result is written into the low float64 element of the destination operand according to the writemask k1. Bits 127:64 of the destination is copied from the corresponding bits of the first source operand (the second operand).

A denormal input value is treated as zero and does not signal #DE, irrespective of MXCSR.DAZ. A denormal result is flushed to zero and does not signal #UE, irrespective of MXCSR.FTZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is ±∞, ±0.0 is returned for that element. Also, if any source element is ±0.0, ±∞ is returned for that element.

The first source operand is an XMM register. The second source operand is an XMM register or a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

A numerically exact implementation of VRCP28xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP28SD ((EVEX Encoded Versions)

```
IF k1[0] OR *no writemask* THEN
        DEST[63: 0] := RCP_28_DP(1.0/SRC2[63: 0]);
ELSE
    IF *merging-masking*            ; merging-masking
        THEN *DEST[63: 0] remains unchanged*
        ELSE                        ; zeroing-masking
            DEST[63: 0] := 0
    FI;
FI;
ENDFOR;
DEST[127:64] := SRC1[127: 64]
DEST[MAXVL-1:128] := 0
```

Table 8-4. VRCP28SD Special Cases

Input Value	Result Value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$0 \le X < 2^{-1022}$	INF	Positive input denormal or zero; #Z
$-2^{-1022} < X \le -0$	-INF	Negative input denormal or zero; #Z
$X > 2^{1022}$	+0.0f	
$X < -2^{1022}$	-0.0f	
$X = +\infty$	+0.0f	
$X = -\infty$	-0.0f	
$X = 2^{-n}$	2^{n}	Exact result (unless input/output is a denormal)
$X = -2^{-n}$	-2^{n}	Exact result (unless input/output is a denormal)

Intel C/C++ Compiler Intrinsic Equivalent

VRCP28SD __m128d _mm_rcp28_round_sd (__m128d a, __m128d b, int sae);
VRCP28SD __m128d _mm_mask_rcp28_round_sd(__m128d s, __mmask8 m, __m128d a, __m128d b, int sae);
VRCP28SD __m128d _mm_maskz_rcp28_round_sd(__mmask8 m, __m128d a, __m128d b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero.

Other Exceptions

See Table 2-49, "Type E3 Class Exception Conditions."

VRCP28PS—Approximation to the Reciprocal of Packed Single Precision Floating-Point Values With Less Than 2^-28 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 CA /r VRCP28PS zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	A	V/V	AVX512ER	Computes the approximate reciprocals (< 2^-28 relative error) of the packed single precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Computes the reciprocal approximation of the float32 values in the source operand (the second operand) and store the results to the destination operand (the first operand) using the writemask k1. The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error prior to final rounding. The final results are rounded to < 2^-23 relative error before written to the destination.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FTZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is ±∞, ±0.0 is returned for that element. Also, if any source element is ±0.0, ±∞ is returned for that element.

The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

A numerically exact implementation of VRCP28xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP28PS (EVEX Encoded Versions)

```
(KL, VL) = (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+31:i] := RCP_28_SP(1.0/SRC[31:0]);
                ELSE DEST[i+31:i] := RCP_28_SP(1.0/SRC[i+31:i]);
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+31:i] := 0
        FI;
    FI;
ENDFOR;
```

Table 8-5. VRCP28PS Special Cases

Input Value	Result Value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$0 \le X < 2^{-126}$	INF	Positive input denormal or zero; #Z
$-2^{-126} < X \le -0$	-INF	Negative input denormal or zero; #Z
$X > 2^{126}$	+0.0f	
$X < -2^{126}$	-0.0f	
$X = +\infty$	+0.0f	
$X = -\infty$	-0.0f	
$X = 2^{-n}$	2^{n}	Exact result (unless input/output is a denormal)
$X = -2^{-n}$	-2^{n}	Exact result (unless input/output is a denormal)

Intel C/C++ Compiler Intrinsic Equivalent

VRCP28PS _mm512_rcp28_round_ps (__m512 a, int sae);
VRCP28PS __m512 _mm512_mask_rcp28_round_ps(__m512 s, __mmask16 m, __m512 a, int sae);
VRCP28PS __m512 _mm512_maskz_rcp28_round_ps(__mmask16 m, __m512 a, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero.

Other Exceptions

See Table 2-48, "Type E2 Class Exception Conditions."

VRCP28SS—Approximation to the Reciprocal of Scalar Single Precision Floating-Point Value With Less Than 2^-28 Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W0 CB /r VRCP28SS xmm1 {k1}{z}, xmm2, xmm3/m32 {sae}	A	V/V	AVX512ER	Computes the approximate reciprocal (< 2^-28 relative error) of the scalar single precision floating-point value in xmm3/m32 and stores the results in xmm1. Under writemask. Also, upper 3 single precision floating-point values (bits[127:32]) from xmm2 is copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Computes the reciprocal approximation of the low float32 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error prior to final rounding. The final result is rounded to < 2^-23 relative error before written into the low float32 element of the destination according to writemask k1. Bits 127:32 of the destination is copied from the corresponding bits of the first source operand (the second operand).

A denormal input value is treated as zero and does not signal #DE, irrespective of MXCSR.DAZ. A denormal result is flushed to zero and does not signal #UE, irrespective of MXCSR.FTZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is ±∞, ±0.0 is returned for that element. Also, if any source element is ±0.0, ±∞ is returned for that element.

The first source operand is an XMM register. The second source operand is an XMM register or a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

A numerically exact implementation of VRCP28xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP28SS ((EVEX Encoded Versions)

```
IF k1[0] OR *no writemask* THEN
        DEST[31: 0] := RCP_28_SP(1.0/SRC2[31: 0]);
ELSE
    IF *merging-masking*            ; merging-masking
        THEN *DEST[31: 0] remains unchanged*
        ELSE                        ; zeroing-masking
            DEST[31: 0] := 0
    FI;
FI;
ENDFOR;
DEST[127:32] := SRC1[127: 32]
DEST[MAXVL-1:128] := 0
```

Table 8-6. VRCP28SS Special Cases

Input Value	Result Value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$0 \le X < 2^{-126}$	INF	Positive input denormal or zero; #Z
$-2^{-126} < X \le -0$	-INF	Negative input denormal or zero; #Z
$X > 2^{126}$	+0.0f	
$X < -2^{126}$	-0.0f	
$X = +\infty$	+0.0f	
$X = -\infty$	-0.0f	
$X = 2^{-n}$	2^{n}	Exact result (unless input/output is a denormal)
$X = -2^{-n}$	-2^{n}	Exact result (unless input/output is a denormal)

Intel C/C++ Compiler Intrinsic Equivalent

VRCP28SS __m128 _mm_rcp28_round_ss (__m128 a, __m128 b, int sae);
VRCP28SS __m128 _mm_mask_rcp28_round_ss(__m128 s, __mmask8 m, __m128 a, __m128 b, int sae);
VRCP28SS __m128 _mm_maskz_rcp28_round_ss(__mmask8 m, __m128 a, __m128 b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero.

Other Exceptions

See Table 2-49, "Type E3 Class Exception Conditions."

VRSQRT28PD—Approximation to the Reciprocal Square Root of Packed Double Precision Floating-Point Values With Less Than 2^-28 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 CC /r VRSQRT28PD zmm1 {k1}{z}, zmm2/m512/m64bcst {sae}	A	V/V	AVX512ER	Computes approximations to the Reciprocal square root (<2^-28 relative error) of the packed double precision floating-point values from zmm2/m512/m64bcst and stores result in zmm1with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Computes the reciprocal square root of the float64 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error.

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as -∞, return the canonical NaN and set the Invalid Flag (#I).

A value of -0 must return -∞ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return -∞ and set the DivByZero flag.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

A numerically exact implementation of VRSQRT28xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT28PD (EVEX Encoded Versions)

```
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+63:i] := (1.0/ SQRT(SRC[63:0]));
                ELSE DEST[i+63:i] := (1.0/ SQRT(SRC[i+63:i]));
            FI;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[i+63:i] := 0
        FI;
    FI;
ENDFOR;
```

Table 8-7. VRSQRT28PD Special Cases

Input Value	Result Value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
X = 2^{-2n}	2^n	
X < 0	QNaN_Indefinite	Including -INF
X = -0 or negative denormal	-INF	#Z
X = +0 or positive denormal	+INF	#Z
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT28PD __m512d _mm512_rsqrt28_round_pd(__m512d a, int sae);
VRSQRT28PD __m512d _mm512_mask_rsqrt28_round_pd(__m512d s, __mmask8 m,__m512d a, int sae);
VRSQRT28PD __m512d _mm512_maskz_rsqrt28_round_pd(__mmask8 m,__m512d a, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero.

Other Exceptions

See Table 2-48, "Type E2 Class Exception Conditions."

VRSQRT28SD—Approximation to the Reciprocal Square Root of Scalar Double Precision Floating-Point Value With Less Than 2^-28 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W1 CD /r VRSQRT28SD xmm1 {k1}{z}, xmm2, xmm3/m64 {sae}	A	V/V	AVX512ER	Computes approximate reciprocal square root (<2^-28 relative error) of the scalar double precision floating-point value from xmm3/m64 and stores result in xmm1with writemask k1. Also, upper double precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Computes the reciprocal square root of the low float64 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal square root is evaluated with less than 2^-28 of maximum relative error. The result is written into the low float64 element of xmm1 according to the writemask k1. Bits 127:64 of the destination is copied from the corresponding bits of the first source operand (the second operand).

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as -∞, return the canonical NaN and set the Invalid Flag (#I).

A value of -0 must return -∞ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return -∞ and set the DivByZero flag.

The first source operand is an XMM register. The second source operand is an XMM register or a 64-bit memory location. The destination operand is a XMM register.

A numerically exact implementation of VRSQRT28xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT28SD (EVEX Encoded Versions)

```
    IF k1[0] OR *no writemask* THEN
                DEST[63: 0] := (1.0/ SQRT(SRC[63: 0]));
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63: 0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63: 0] := 0
        FI;
    FI;
ENDFOR;
DEST[127:64] := SRC1[127: 64]
DEST[MAXVL-1:128] := 0
```

Table 8-8. VRSQRT28SD Special Cases

Input Value	Result Value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
X = 2^{-2n}	2^{n}	
X < 0	QNaN_Indefinite	Including -INF
X = -0 or negative denormal	-INF	#Z
X = +0 or positive denormal	+INF	#Z
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT28SD __m128d _mm_rsqrt28_round_sd(__m128d a, __m128d b, int rounding);
VRSQRT28SD __m128d _mm_mask_rsqrt28_round_sd(__m128d s, __mmask8 m,__m128d a, __m128d b, int rounding);
VRSQRT28SD __m128d _mm_maskz_rsqrt28_round_sd(__mmask8 m,__m128d a, __m128d b, int rounding);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero.

Other Exceptions

See Table 2-49, "Type E3 Class Exception Conditions."

VRSQRT28PS—Approximation to the Reciprocal Square Root of Packed Single Precision Floating-Point Values With Less Than 2^-28 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 CC /r VRSQRT28PS zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	A	V/V	AVX512ER	Computes approximations to the Reciprocal square root (<2^-28 relative error) of the packed single precision floating-point values from zmm2/m512/m32bcst and stores result in zmm1with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Computes the reciprocal square root of the float32 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error prior to final rounding. The final results is rounded to < 2^-23 relative error before written to the destination.

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as -∞, return the canonical NaN and set the Invalid Flag (#I).

A value of -0 must return -∞ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return -∞ and set the DivByZero flag.

The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

A numerically exact implementation of VRSQRT28xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT28PS (EVEX Encoded Versions)

```
(KL, VL) = (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+31:i] := (1.0/ SQRT(SRC[31:0]));
                ELSE DEST[i+31:i] := (1.0/ SQRT(SRC[i+31:i]));
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+31:i] := 0
        FI;
    FI;
ENDFOR;
```

Table 8-9. VRSQRT28PS Special Cases

Input Value	Result Value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
X = 2^{-2n}	2^{n}	
X < 0	QNaN_Indefinite	Including -INF
X = -0 or negative denormal	-INF	#Z
X = +0 or positive denormal	+INF	#Z
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT28PS __m512 _mm512_rsqrt28_round_ps(__m512 a, int sae);
VRSQRT28PS __m512 _mm512_mask_rsqrt28_round_ps(__m512 s, __mmask16 m,__m512 a, int sae);
VRSQRT28PS __m512 _mm512_maskz_rsqrt28_round_ps(__mmask16 m,__m512 a, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero.

Other Exceptions

See Table 2-48, "Type E2 Class Exception Conditions."

VRSQRT28SS—Approximation to the Reciprocal Square Root of Scalar Single Precision Floating-Point Value With Less Than 2^-28 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W0 CD /r VRSQRT28SS xmm1 {k1}{z}, xmm2, xmm3/m32 {sae}	A	V/V	AVX512ER	Computes approximate reciprocal square root (<2^-28 relative error) of the scalar single precision floating-point value from xmm3/m32 and stores result in xmm1with writemask k1. Also, upper 3 single precision floating-point value (bits[127:32]) from xmm2 is copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Computes the reciprocal square root of the low float32 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal square root is evaluated with less than 2^-28 of maximum relative error prior to final rounding. The final result is rounded to < 2^-23 relative error before written to the low float32 element of the destination according to the writemask k1. Bits 127:32 of the destination is copied from the corresponding bits of the first source operand (the second operand).

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as -∞, return the canonical NaN and set the Invalid Flag (#I).

A value of -0 must return -∞ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return -∞ and set the DivByZero flag.

The first source operand is an XMM register. The second source operand is an XMM register or a 32-bit memory location. The destination operand is a XMM register.

A numerically exact implementation of VRSQRT28xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT28SS (EVEX Encoded Versions)

```
    IF k1[0] OR *no writemask* THEN
                DEST[31: 0] := (1.0/ SQRT(SRC[31: 0]));
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31: 0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31: 0] := 0
        FI;
    FI;
ENDFOR;
DEST[127:32] := SRC1[127: 32]
DEST[MAXVL-1:128] := 0
```

Table 8-10. VRSQRT28SS Special Cases

Input Value	Result Value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$X = 2^{-2n}$	2^n	
X < 0	QNaN_Indefinite	Including -INF
X = -0 or negative denormal	-INF	#Z
X = +0 or positive denormal	+INF	#Z
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT28SS __m128 _mm_rsqrt28_round_ss(__m128 a, __m128 b, int rounding);
VRSQRT28SS __m128 _mm_mask_rsqrt28_round_ss(__m128 s, __mmask8 m,__m128 a,__m128 b, int rounding);
VRSQRT28SS __m128 _mm_maskz_rsqrt28_round_ss(__mmask8 m,__m128 a,__m128 b, int rounding);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero.

Other Exceptions

See Table 2-49, "Type E3 Class Exception Conditions."

VSCATTERPF0DPS/VSCATTERPF0QPS/VSCATTERPF0DPD/VSCATTERPF0QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T0 Hint With Intent to Write

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C6 /5 /vsib VSCATTERPF0DPS vm32z {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing single precision data using writemask k1 and T0 hint with intent to write.
EVEX.512.66.0F38.W0 C7 /5 /vsib VSCATTERPF0QPS vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing single precision data using writemask k1 and T0 hint with intent to write.
EVEX.512.66.0F38.W1 C6 /5 /vsib VSCATTERPF0DPD vm32y {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing double precision data using writemask k1 and T0 hint with intent to write.
EVEX.512.66.0F38.W1 C7 /5 /vsib VSCATTERPF0QPD vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing double precision data using writemask k1 and T0 hint with intent to write.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	N/A	N/A	N/A

Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

cache lines will be brought into exclusive state (RFO) specified by a locality hint (T0):

• T0 (temporal data)—prefetch data into the first level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

(1) The prefetches may happen in any order (or not at all). The instruction is a hint.

(2) The mask is left unchanged.

(3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.

(4) No FP nor memory faults may be produced by this instruction.

(5) Prefetches do not handle cache line splits

(6) A #UD is signaled if the memory operand is encoded without the SIB byte.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist.
VINDEX stands for the memory operand vector of indices (a vector register).
SCALE stands for the memory operand scalar (1, 2, 4 or 8).
DISP is the optional 1, 2 or 4 byte displacement.
PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

VSCATTERPF0DPS (EVEX Encoded Version)

```
(KL, VL) = (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR
```

VSCATTERPF0DPD (EVEX Encoded Version)

```
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR
```

VSCATTERPF0QPS (EVEX Encoded Version)

```
(KL, VL) = (8, 256)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR
```

VSCATTERPF0QPD (EVEX Encoded Version)

```
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalent

VSCATTERPF0DPD void _mm512_prefetch_i32scatter_pd(void *base, __m256i vdx, int scale, int hint);
VSCATTERPF0DPD void _mm512_mask_prefetch_i32scatter_pd(void *base, __mmask8 m, __m256i vdx, int scale, int hint);
VSCATTERPF0DPS void _mm512_prefetch_i32scatter_ps(void *base, __m512i vdx, int scale, int hint);
VSCATTERPF0DPS void _mm512_mask_prefetch_i32scatter_ps(void *base, __mmask16 m, __m512i vdx, int scale, int hint);
VSCATTERPF0QPD void _mm512_prefetch_i64scatter_pd(void * base, __m512i vdx, int scale, int hint);
VSCATTERPF0QPD void _mm512_mask_prefetch_i64scatter_pd(void * base, __mmask8 m, __m512i vdx, int scale, int hint);
VSCATTERPF0QPS void _mm512_prefetch_i64scatter_ps(void * base, __m512i vdx, int scale, int hint);
VSCATTERPF0QPS void _mm512_mask_prefetch_i64scatter_ps(void * base, __mmask8 m, __m512i vdx, int scale, int hint);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-64, "Type E12NP Class Exception Conditions."

VSCATTERPF1DPS/VSCATTERPF1QPS/VSCATTERPF1DPD/VSCATTERPF1QPD—Sparse Prefetch Packed SP/DP Data Values With Signed Dword, Signed Qword Indices Using T1 Hint With Intent to Write

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C6 /6 /vsib VSCATTERPF1DPS vm32z {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing single precision data using writemask k1 and T1 hint with intent to write.
EVEX.512.66.0F38.W0 C7 /6 /vsib VSCATTERPF1QPS vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing single precision data using writemask k1 and T1 hint with intent to write.
EVEX.512.66.0F38.W1 C6 /6 /vsib VSCATTERPF1DPD vm32y {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing double precision data using writemask k1 and T1 hint with intent to write.
EVEX.512.66.0F38.W1 C7 /6 /vsib VSCATTERPF1QPD vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing double precision data using writemask k1 and T1 hint with intent to write.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	N/A	N/A	N/A

Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

cache lines will be brought into exclusive state (RFO) specified by a locality hint (T1):

• T1 (temporal data)—prefetch data into the second level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

(1) The prefetches may happen in any order (or not at all). The instruction is a hint.

(2) The mask is left unchanged.

(3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.

(4) No FP nor memory faults may be produced by this instruction.

(5) Prefetches do not handle cache line splits

(6) A #UD is signaled if the memory operand is encoded without the SIB byte.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist.
VINDEX stands for the memory operand vector of indices (a vector register).
SCALE stands for the memory operand scalar (1, 2, 4 or 8).
DISP is the optional 1, 2 or 4 byte displacement.
PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

VSCATTERPF1DPS (EVEX Encoded Version)

```
(KL, VL) = (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=1, RFO = 1)
    FI;
ENDFOR
```

VSCATTERPF1DPD (EVEX Encoded Version)

```
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=1, RFO = 1)
    FI;
ENDFOR
```

VSCATTERPF1QPS (EVEX Encoded Version)

```
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=1, RFO = 1)
    FI;
ENDFOR
```

VSCATTERPF1QPD (EVEX Encoded Version)

```
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=1, RFO = 1)
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalent

VSCATTERPF1DPD void _mm512_prefetch_i32scatter_pd(void *base, __m256i vdx, int scale, int hint);
VSCATTERPF1DPD void _mm512_mask_prefetch_i32scatter_pd(void *base, __mmask8 m, __m256i vdx, int scale, int hint);
VSCATTERPF1DPS void _mm512_prefetch_i32scatter_ps(void *base, __m512i vdx, int scale, int hint);
VSCATTERPF1DPS void _mm512_mask_prefetch_i32scatter_ps(void *base, __mmask16 m, __m512i vdx, int scale, int hint);
VSCATTERPF1QPD void _mm512_prefetch_i64scatter_pd(void * base, __m512i vdx, int scale, int hint);
VSCATTERPF1QPD void _mm512_mask_prefetch_i64scatter_pd(void * base, __mmask8 m, __m512i vdx, int scale, int hint);
VSCATTERPF1QPS void _mm512_prefetch_i64scatter_ps(void *base, __m512i vdx, int scale, int hint);
VSCATTERPF1QPS void _mm512_mask_prefetch_i64scatter_ps(void *base, __mmask8 m, __m512i vdx, int scale, int hint);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-64, "Type E12NP Class Exception Conditions."

APPENDIX A
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 A-2 through Table A-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 A-7 through Table A-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.

A.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. Cells marked "Reserved-NOP" are also reserved but may behave as NOP on certain processors. Software should not use opcodes corresponding blank cells or cells marked "Reserved-NOP" nor depend on the current behavior of those opcodes.

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 A-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 A-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 A-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 A.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 A-2 and Table A-3, see Section A.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 A.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.

A.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.

A.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).

A.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.

A.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).

A.2.4 Opcode Look-up Examples for One, Two, and Three-Byte Opcodes

This section provides examples that demonstrate how opcode maps are used.

A.2.4.1 One-Byte Opcode Instructions

The opcode map for 1-byte opcodes is shown in Table A-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 A.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 A.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 A.2.
- A ModR/M byte is required and includes an opcode extension in the reg field in the ModR/M byte. Use Table A-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 A-1. Look-up Example for 1-Byte Opcodes

Opcode 030500000000H for an ADD instruction is interpreted using the 1-byte opcode map (Table A-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 A.4).

A.2.4.2 Two-Byte Opcode Instructions

The two-byte opcode map shown in Table A-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 A-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 A-3 (except when the second opcode byte is the 3-byte escape opcodes 38H or 3AH; in this situation refer to Section A.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 A.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 A.2.
- A ModR/M byte is required and includes an opcode extension in the reg field in the ModR/M byte. Use Table A-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 A-2. Look-up Example for 2-Byte Opcodes

Look-up opcode 0FA4050000000003H for a SHLD instruction using Table A-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.

A.2.4.3 Three-Byte Opcode Instructions

The three-byte opcode maps shown in Table A-4 and Table A-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 A-4 or Table A-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 A-4 or Table A-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 A.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 A.2.

Example A-3. Look-up Example for 3-Byte Opcodes

Look-up opcode 660F3A0FC108H for a PALIGNR instruction using Table A-5.

- 66H is a prefix and 0F3AH indicate to use Table A-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.

A.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.

A.2.5 Superscripts Utilized in Opcode Tables

Table A-1 contains notes on particular encodings. These notes are indicated in the following opcode maps by superscripts. Gray cells indicate instruction groupings.

Table A-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 A.4, "Opcode Extensions For One-Byte And Two-byte Opcodes").
1B	Use the 0F0B opcode (UD2 instruction), the 0FB9H opcode (UD1 instruction), the 0FFFH opcode (UD0 instruction), or the D6 opcode (UDB instruction) 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 A-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).
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.

A.3 ONE, TWO, AND THREE-BYTE OPCODE MAPS

See Table A-2 through Table A-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 A-2. One-byte Opcode Map: (00H — F7H) *

	0	1	2	3	4	5	6	7
0	ADD						PUSH ES^{i64}	POP ES^{i64}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
1	ADC						PUSH SS^{i64}	POP SS^{i64}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
2	AND						SEG=ES (Prefix)	DAA^{i64}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
3	XOR						SEG=SS (Prefix)	AAA^{i64}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
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}				TEST		XCHG	
	Eb, Ib	Ev, Iz	Eb, Ib^{i64}	Ev, Ib	Eb, Gb	Ev, Gv	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				MOVS/B Yb, Xb	MOVS/W/D/Q Yv, Xv	CMPS/B Xb, Yb	CMPS/W/D/Q Xv, Yv
	AL, Ob	rAX, Ov	Ob, AL	Ov, rAX				
B	MOV immediate byte into byte register							
	AL/R8B, Ib	CL/R9B, Ib	DL/R10B, Ib	BL/R11B, Ib	AH/R12B, Ib	CH/R13B, Ib	DH/R14B, Ib	BH/R15B, Ib
C	Shift Grp 2^{1A}		near RET^{f64} Iw	near RET^{f64}	LES^{i64} Gz, Mp VEX+2byte	LDS^{i64} Gz, Mp VEX+1byte	Grp 11^{1A} - MOV	
	Eb, Ib	Ev, Ib					Eb, Ib	Ev, Iz
D	Shift Grp 2^{1A}				AAM^{i64} Ib	AAD^{i64} Ib	SALC^{i64} UDB^{o64,1B}	XLAT/ XLATB
	Eb, 1	Ev, 1	Eb, CL	Ev, CL				
E	LOOPNE^{f64}/ LOOPNZ^{f64} Jb	LOOPE^{f64}/ LOOPZ^{f64} Jb	LOOP^{f64} Jb	JrCXZ^{f64}/ Jb	IN		OUT	
					AL, Ib	eAX, Ib	Ib, AL	Ib, eAX
F	LOCK (Prefix)	INT1	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						PUSH CS[i64]	2-byte escape (Table A-3)
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
1	SBB						PUSH DS[i64]	POP DS[i64]
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
2	SUB						SEG=CS (Prefix)	DAS[i64]
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
3	CMP						SEG=DS (Prefix)	AAS[i64]
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
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				MOV Ev, Sw	LEA Gv, M	MOV Sw, Ew	Grp 1A[1A] POP[d64] Ev
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev				
9	CBW/ CWDE/ CDQE	CWD/ CDQ/ CQO	far CALL[i64] Ap	FWAIT/ WAIT	PUSHF/D/Q [d64]/ Fv	POPF/D/Q [d64]/ Fv	SAHF	LAHF
A	TEST		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, Yv
	AL, Ib	rAX, Iz						
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]	far RET Iw	far RET	INT3	INT Ib	INTO[i64]	IRET/D/Q
D	ESC (Escape to coprocessor instruction set)							
E	near CALL[f64] Jz	JMP			IN		OUT	
		near[f64] Jz	far[i64] Ap	short[f64] Jb	AL, DX	eAX, DX	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 A-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 Vps, Wps	vmovups Wps, Vps	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 Vpd, Wpd	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		
	F3		WBNOINVD						
1		Prefetch[1C] (Grp 16[1A])	Reserved-NOP	bndldx	bndstx	(Grp 18[1A])	Reserved-NOP		NOP /0 Ev
	66			bndmov	bndmov				
	F3			bndcl	bndmk				
	F2			bndcu	bndcn				

Table A-3. Two-byte Opcode Map: 08H — 7FH (First Byte is 0FH) *

	pfx	8	9	A	B	C	D	E	F
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	UD0[1B]
	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 A-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, Wx	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		sha1nexte Vdq,Wdq	sha1msg1 Vdq,Wdq	sha1msg2 Vdq,Wdq	sha256rnds2 Vdq,Wdq	sha256msg1 Vdq,Wdq	sha256msg2 Vdq,Wdq		
	66								
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	UWRMSR							
	F2	URDMSR							
	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 A-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						sha1rnds4 Vdq,Wdq,Ib			
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.

A.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 A-1) as an extension of the opcode.

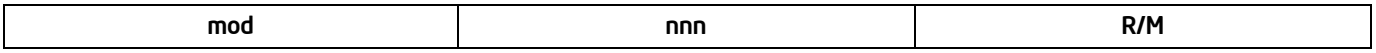


Figure A-1. ModR/M Byte nnn Field (Bits 5, 4, and 3)

Opcodes that have opcode extensions are indicated in Table A-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.

A.4.1 Opcode Look-up Examples Using Opcode Extensions

An Example is provided below.

Example A-4. Interpreting an ADD Instruction

An ADD instruction with a 1-byte opcode of 80H is a Group 1 instruction:

- Table A-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 A-5. Looking Up 0F01C3H

Look up opcode 0F01C3 for a VMRESUME instruction by using Table A-2, Table A-3, and Table A-6:

- 0F indicates that this instruction is in the 2-byte opcode map.
- 01 (row 0, column 1 in Table A-3) reveals that this opcode is in Group 7 of Table A-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 A-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.

A.4.2 Opcode Extension Tables

See Table A-6 below.

Table A-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	near CALL^{f64} Ev	far CALL Ep	near JMP^{f64} Ev	far JMP Mp	PUSH^{d64} Ev	
0F 00	6	mem, 11B		SLDT Rv/Mw	STR Rv/Mw	LLDT Ew	LTR Ew	VERR Ew	VERW Ew		
			F2							LKGS^{o64} Ew	
0F 01	7	mem		SGDT Ms	SIDT Ms	LGDT Ms	LIDT Ms	SMSW Mw/Rv		LMSW Ew	INVLPG Mb
			F3						RSTORSSP Mq		
		11B		VMCALL (001) VMLAUNCH (010) VMRESUME (011) VMXOFF (100) PCONFIG (101) WRMSRNS (110) PBNDKB (111)	MONITOR (000) MWAIT (001) CLAC (010) STAC (011) ENCLS (111)	XGETBV (000) XSETBV (001) VMFUNC (100) XEND (101) XTEST (110) ENCLU(111)			SERIALIZE (000) RDPKRU (110) WRPKRU (111)		SWAPGS^{o64}(000) RDTSCP (001)
			66		TDCALL (100) SEAMRET (101) SEAMOPS (110) SEAMCALL^{o64} (111)						
			F2	RDMSRLIST (110)^{o64}	ERETS^{o64} (010)				XSUSLDTRK (000) XRESLDTRK (001)		
			F3	WRMSRLIST (110)^{o64}	ERETU^{o64} (010)				SETSSBSY (000) SAVEPREVSSP (010) UIRET (100) CLUI (110) STUI (111)		
0F BA	8	mem, 11B						BT	BTS	BTR	BTC
0F C7	9	mem			CMPXCH8B Mq CMPXCHG16B Mdq		XRSTORS	XSAVEC	XSAVES	VMPTRLD Mq	VMPTRST Mq
			66							VMCLEAR Mq	
			F3							VMXON Mq	
		11B								RDRAND Rv	RDSEED Rv
			F3							SENDUIPI^{o64} Rq	RDPID Rd/q
0F B9	10	mem		UD1^{1B}							
		11B									
C6	11	mem		MOV Eb, Ib							
		11B									XABORT (000) Ib
C7		mem		MOV Ev, Iz							
		11B									XBEGIN (000) Jz

Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number * (Contd.)

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

Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number * (Contd.)

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 AE	15	mem		FXSAVE	FXRSTOR	LDMXCSR	STMXCSR	XSAVE	XRSTOR	XSAVEOPT	CLFLUSH
			66							CLWB	CLFLUSHOPT
			F3					PTWRITE My		CLRSSBSY	
		11B							LFENCE (000)	MFENCE (000)	SFENCE (000)
			66							TPAUSE Rd	
			F2							UMWAIT Rd	
			F3	RDFSBASE Ry	RDGSBASE Ry	WRFSBASE Ry	WRGSBASE Ry	PTWRITE Ry	INCSSP	UMONITOR Rv	
0F 18	16			prefetch NTA	prefetch T0	prefetch T1	prefetch T2	Reserved NOP			
		11B		Reserved NOP							
VEX.0F38 F3	17	mem			BLSR[v] By, Ey	BLSMSK[v] By, Ey	BLSI[v] By, Ey				
		11B									
0F 1C	18	mem		CLDEMOTE	Reserved NOP						
			66, F2, F3	Reserved NOP							
		11B	any								

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.

A.5 ESCAPE OPCODE INSTRUCTIONS

Opcode maps for coprocessor escape instruction opcodes (x87 floating-point instruction opcodes) are in Table A-7 through Table A-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 A.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.

A.5.1 Opcode Look-up Examples for Escape Instruction Opcodes

Examples are provided below.

Example A-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 A-9).
- The double-real value to be loaded is at 00000004H (the 32-bit displacement that follows and belongs to this opcode).

Example A-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 A.4.
- In Table A-8, the ModR/M byte C1H indicates row C, column 1 (the FADD instruction using ST(0), ST(1) as operands).

A.5.2 Escape Opcode Instruction Tables

Tables are listed below.

A.5.2.1 Escape Opcodes with D8 as First Byte

Table A-7 and A-8 contain maps for the escape instruction opcodes that begin with D8H. Table A-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 A-1) selects the instruction.

Table A-7. D8 Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte (refer to Figure A.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 A-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 A-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.

A.5.2.2 Escape Opcodes with D9 as First Byte

Table A-9 and A-10 contain maps for escape instruction opcodes that begin with D9H. Table A-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 A-1) selects the instruction.

.

Table A-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 A-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 A-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.

A.5.2.3 Escape Opcodes with DA as First Byte

Table A-11 and A-12 contain maps for escape instruction opcodes that begin with DAH. Table A-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 A-1) selects the instruction.

Table A-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 A-12 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 A-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.

A.5.2.4 Escape Opcodes with DB as First Byte

Table A-13 and A-14 contain maps for escape instruction opcodes that begin with DBH. Table A-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 A-1) selects the instruction.

Table A-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 A-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 A-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.

A.5.2.5 Escape Opcodes with DC as First Byte

Table A-15 and A-16 contain maps for escape instruction opcodes that begin with DCH. Table A-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 A-1) selects the instruction.

Table A-15. DC Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte (refer to Figure A-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 A-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 A-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.

A.5.2.6 Escape Opcodes with DD as First Byte

Table A-17 and A-18 contain maps for escape instruction opcodes that begin with DDH. Table A-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 A-1) selects the instruction.

Table A-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 A-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 A-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.

A.5.2.7 Escape Opcodes with DE as First Byte

Table A-19 and A-20 contain opcode maps for escape instruction opcodes that begin with DEH. Table A-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 A-1) selects the instruction.

Table A-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 A-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 A-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.

A.5.2.8 Escape Opcodes with DF As First Byte

Table A-21 and A-22 contain the opcode maps for escape instruction opcodes that begin with DFH. Table A-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 A-1) selects the instruction.

Table A-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 A-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 A-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.

APPENDIX B
INSTRUCTION FORMATS AND ENCODINGS

This appendix provides machine instruction formats and encodings of IA-32 instructions. The first section describes the IA-32 architecture's machine instruction format. The remaining sections show the formats and encoding of general-purpose, MMX, P6 family, SSE/SSE2/SSE3, x87 FPU instructions, and VMX instructions. Those instruction formats also apply to Intel 64 architecture. Instruction formats used in 64-bit mode are provided as supersets of the above.

B.1 MACHINE INSTRUCTION FORMAT

All Intel Architecture instructions are encoded using subsets of the general machine instruction format shown in Figure B-1. Each instruction consists of:

- an opcode
- a register and/or address mode specifier consisting of the ModR/M byte and sometimes the scale-index-base (SIB) byte (if required)
- a displacement and an immediate data field (if required)

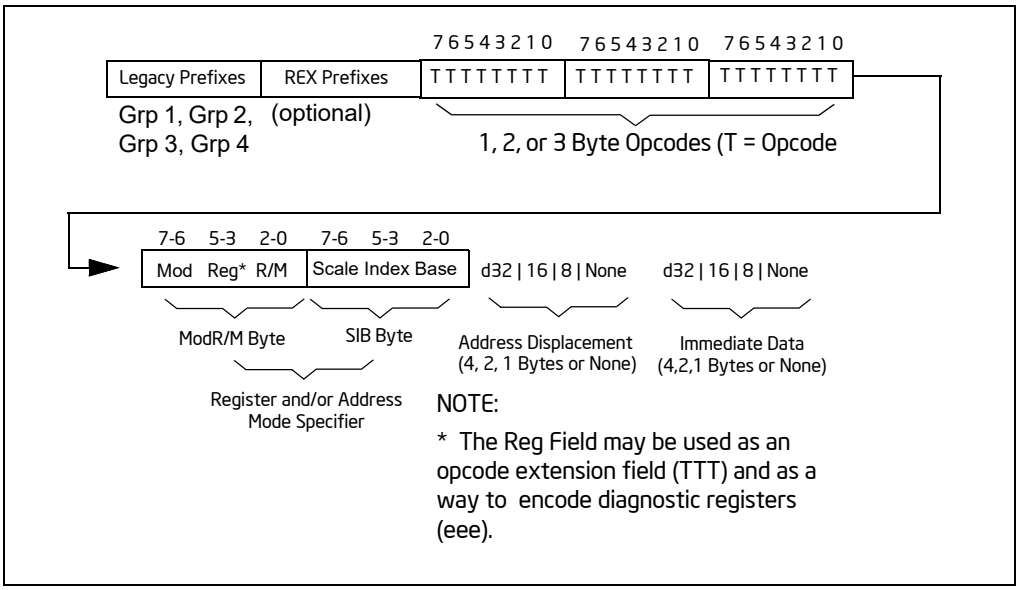


Figure B-1. General Machine Instruction Format

The following sections discuss this format.

B.1.1 Legacy Prefixes

The legacy prefixes noted in Figure B-1 include 66H, 67H, F2H, and F3H. They are optional, except when F2H, F3H, and 66H are used in instruction extensions. Legacy prefixes must be placed before REX prefixes.

Refer to Chapter 2, "Instruction Format," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, for more information on legacy prefixes.

B.1.2 REX Prefixes

REX prefixes are a set of 16 opcodes that span one row of the opcode map and occupy entries 40H to 4FH. These opcodes represent valid instructions (INC or DEC) in IA-32 operating modes and in compatibility mode. In 64-bit mode, the same opcodes represent the instruction prefix REX and are not treated as individual instructions.

Refer to Chapter 2, "Instruction Format," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, for more information on REX prefixes.

B.1.3 Opcode Fields

The primary opcode for an instruction is encoded in one to three bytes of the instruction. Within the primary opcode, smaller encoding fields may be defined. These fields vary according to the class of operation being performed.

Almost all instructions that refer to a register and/or memory operand have a register and/or address mode byte following the opcode. This byte, the ModR/M byte, consists of the mod field (2 bits), the reg field (3 bits; this field is sometimes an opcode extension), and the R/M field (3 bits). Certain encodings of the ModR/M byte indicate that a second address mode byte, the SIB byte, must be used.

If the addressing mode specifies a displacement, the displacement value is placed immediately following the ModR/M byte or SIB byte. Possible sizes are 8, 16, or 32 bits. If the instruction specifies an immediate value, the immediate value follows any displacement bytes. The immediate, if specified, is always the last field of the instruction.

Refer to Chapter 2, "Instruction Format," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, for more information on opcodes.

B.1.4 Special Fields

Table B-1 lists bit fields that appear in certain instructions, sometimes within the opcode bytes. All of these fields (except the d bit) occur in the general-purpose instruction formats in Table B-13.

Table B-1. Special Fields Within Instruction Encodings

Field Name	Description	Number of Bits
reg	General-register specifier (see Table B-4 or B-5).	3
w	Specifies if data is byte or full-sized, where full-sized is 16 or 32 bits (see Table B-6).	1
s	Specifies sign extension of an immediate field (see Table B-7).	1
sreg2	Segment register specifier for CS, SS, DS, ES (see Table B-8).	2
sreg3	Segment register specifier for CS, SS, DS, ES, FS, GS (see Table B-8).	3
eee	Specifies a special-purpose (control or debug) register (see Table B-9).	3
tttn	For conditional instructions, specifies a condition asserted or negated (see Table B-12).	4
d	Specifies direction of data operation (see Table B-11).	1

B.1.4.1 Reg Field (reg) for Non-64-Bit Modes

The reg field in the ModR/M byte specifies a general-purpose register operand. The group of registers specified is modified by the presence and state of the w bit in an encoding (refer to Section B.1.4.3). Table B-2 shows the encoding of the reg field when the w bit is not present in an encoding; Table B-3 shows the encoding of the reg field when the w bit is present.

Table B-2. Encoding of reg Field When w Field is Not Present in Instruction

reg Field	Register Selected during 16-Bit Data Operations	Register Selected during 32-Bit Data Operations
000	AX	EAX
001	CX	ECX
010	DX	EDX
011	BX	EBX
100	SP	ESP
101	BP	EBP
110	SI	ESI
111	DI	EDI

Table B-3. Encoding of reg Field When w Field is Present in Instruction

Register Specified by reg Field During 16-Bit Data Operations		
reg	Function of w Field	
	When w = 0	When w = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Register Specified by reg Field During 32-Bit Data Operations		
reg	Function of w Field	
	When w = 0	When w = 1
000	AL	EAX
001	CL	ECX
010	DL	EDX
011	BL	EBX
100	AH	ESP
101	CH	EBP
110	DH	ESI
111	BH	EDI

B.1.4.2 Reg Field (reg) for 64-Bit Mode

Just like in non-64-bit modes, the reg field in the ModR/M byte specifies a general-purpose register operand. The group of registers specified is modified by the presence of and state of the w bit in an encoding (refer to Section B.1.4.3). Table B-4 shows the encoding of the reg field when the w bit is not present in an encoding; Table B-5 shows the encoding of the reg field when the w bit is present.

Table B-4. Encoding of reg Field When w Field is Not Present in Instruction

reg Field	Register Selected during 16-Bit Data Operations	Register Selected during 32-Bit Data Operations	Register Selected during 64-Bit Data Operations
000	AX	EAX	RAX
001	CX	ECX	RCX
010	DX	EDX	RDX
011	BX	EBX	RBX
100	SP	ESP	RSP
101	BP	EBP	RBP
110	SI	ESI	RSI
111	DI	EDI	RDI

Table B-5. Encoding of reg Field When w Field is Present in Instruction

Register Specified by reg Field During 16-Bit Data Operations		
reg	Function of w Field	
	When w = 0	When w = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH[1]	SP
101	CH[1]	BP
110	DH[1]	SI
111	BH[1]	DI

Register Specified by reg Field During 32-Bit Data Operations		
reg	Function of w Field	
	When w = 0	When w = 1
000	AL	EAX
001	CL	ECX
010	DL	EDX
011	BL	EBX
100	AH*	ESP
101	CH*	EBP
110	DH*	ESI
111	BH*	EDI

NOTES:

1. AH, CH, DH, BH can not be encoded when REX prefix is used. Such an expression defaults to the low byte.

B.1.4.3 Encoding of Operand Size (w) Bit

The current operand-size attribute determines whether the processor is performing 16-bit, 32-bit or 64-bit operations. Within the constraints of the current operand-size attribute, the operand-size bit (w) can be used to indicate operations on 8-bit operands or the full operand size specified with the operand-size attribute. Table B-6 shows the encoding of the w bit depending on the current operand-size attribute.

Table B-6. Encoding of Operand Size (w) Bit

w Bit	Operand Size When Operand-Size Attribute is 16 Bits	Operand Size When Operand-Size Attribute is 32 Bits
0	8 Bits	8 Bits
1	16 Bits	32 Bits

B.1.4.4 Sign-Extend (s) Bit

The sign-extend (s) bit occurs in instructions with immediate data fields that are being extended from 8 bits to 16 or 32 bits. See Table B-7.

Table B-7. Encoding of Sign-Extend (s) Bit

s	Effect on 8-Bit Immediate Data	Effect on 16- or 32-Bit Immediate Data
0	None	None
1	Sign-extend to fill 16-bit or 32-bit destination	None

B.1.4.5 Segment Register (sreg) Field

When an instruction operates on a segment register, the reg field in the ModR/M byte is called the sreg field and is used to specify the segment register. Table B-8 shows the encoding of the sreg field. This field is sometimes a 2-bit field (sreg2) and other times a 3-bit field (sreg3).

Table B-8. Encoding of the Segment Register (sreg) Field

2-Bit sreg2 Field	Segment Register Selected
00	ES
01	CS
10	SS
11	DS

3-Bit sreg3 Field	Segment Register Selected
000	ES
001	CS
010	SS
011	DS
100	FS
101	GS
110	Reserved[1]
111	Reserved

NOTES:

1. Do not use reserved encodings.

B.1.4.6 Special-Purpose Register (eee) Field

When control or debug registers are referenced in an instruction they are encoded in the eee field, located in bits 5 though 3 of the ModR/M byte (an alternate encoding of the sreg field). See Table B-9.

Table B-9. Encoding of Special-Purpose Register (eee) Field

eee	Control Register	Debug Register
000	CR0	DR0
001	Reserved[1]	DR1
010	CR2	DR2
011	CR3	DR3
100	CR4	Reserved
101	Reserved	Reserved
110	Reserved	DR6
111	Reserved	DR7

NOTES:

1. Do not use reserved encodings.

B.1.4.7 Condition Test (tttn) Field

For conditional instructions (such as conditional jumps and set on condition), the condition test field (tttn) is encoded for the condition being tested. The tt part of the field gives the condition to test and the n part indicates whether to use the condition (n = 0) or its negation (n = 1).

- For 1-byte primary opcodes, the tttn field is located in bits 3, 2, 1, and 0 of the opcode byte.
- For 2-byte primary opcodes, the tttn field is located in bits 3, 2, 1, and 0 of the second opcode byte.

Table B-10 shows the encoding of the tttn field.

Table B-10. Encoding of Conditional Test (tttn) Field

t t t n	Mnemonic	Condition
0000	O	Overflow
0001	NO	No overflow
0010	B, NAE	Below, Not above or equal
0011	NB, AE	Not below, Above or equal
0100	E, Z	Equal, Zero
0101	NE, NZ	Not equal, Not zero
0110	BE, NA	Below or equal, Not above
0111	NBE, A	Not below or equal, Above
1000	S	Sign
1001	NS	Not sign
1010	P, PE	Parity, Parity Even
1011	NP, PO	Not parity, Parity Odd
1100	L, NGE	Less than, Not greater than or equal to
1101	NL, GE	Not less than, Greater than or equal to
1110	LE, NG	Less than or equal to, Not greater than
1111	NLE, G	Not less than or equal to, Greater than

B.1.4.8 Direction (d) Bit

In many two-operand instructions, a direction bit (d) indicates which operand is considered the source and which is the destination. See Table B-11.

- When used for integer instructions, the d bit is located at bit 1 of a 1-byte primary opcode. Note that this bit does not appear as the symbol "d" in Table B-13; the actual encoding of the bit as 1 or 0 is given.
- When used for floating-point instructions (in Table B-16), the d bit is shown as bit 2 of the first byte of the primary opcode.

Table B-11. Encoding of Operation Direction (d) Bit

d	Source	Destination
0	reg Field	ModR/M or SIB Byte
1	ModR/M or SIB Byte	reg Field

B.1.5 Other Notes

Table B-12 contains notes on particular encodings. These notes are indicated in the tables shown in the following sections by superscripts.

Table B-12. Notes on Instruction Encoding

Symbol	Note
A	A value of 11B in bits 7 and 6 of the ModR/M byte is reserved.
B	A value of 01B (or 10B) in bits 7 and 6 of the ModR/M byte is reserved.

B.2 GENERAL-PURPOSE INSTRUCTION FORMATS AND ENCODINGS FOR NON-64-BIT MODES

Table B-13 shows machine instruction formats and encodings for general purpose instructions in non-64-bit modes.

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes

Instruction and Format	Encoding
AAA – ASCII Adjust after Addition	0011 0111
AAD – ASCII Adjust AX before Division	1101 0101 : 0000 1010
AAM – ASCII Adjust AX after Multiply	1101 0100 : 0000 1010
AAS – ASCII Adjust AL after Subtraction	0011 1111
ADC – ADD with Carry	
register1 to register2	0001 000w : 11 reg1 reg2
register2 to register1	0001 001w : 11 reg1 reg2
memory to register	0001 001w : mod reg r/m
register to memory	0001 000w : mod reg r/m
immediate to register	1000 00sw : 11 010 reg : immediate data
immediate to AL, AX, or EAX	0001 010w : immediate data
immediate to memory	1000 00sw : mod 010 r/m : immediate data
ADD – Add	
register1 to register2	0000 000w : 11 reg1 reg2
register2 to register1	0000 001w : 11 reg1 reg2
memory to register	0000 001w : mod reg r/m
register to memory	0000 000w : mod reg r/m
immediate to register	1000 00sw : 11 000 reg : immediate data
immediate to AL, AX, or EAX	0000 010w : immediate data
immediate to memory	1000 00sw : mod 000 r/m : immediate data
AND – Logical AND	
register1 to register2	0010 000w : 11 reg1 reg2
register2 to register1	0010 001w : 11 reg1 reg2
memory to register	0010 001w : mod reg r/m
register to memory	0010 000w : mod reg r/m
immediate to register	1000 00sw : 11 100 reg : immediate data
immediate to AL, AX, or EAX	0010 010w : immediate data
immediate to memory	1000 00sw : mod 100 r/m : immediate data

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

Instruction and Format	Encoding
ARPL – Adjust RPL Field of Selector	
from register	0110 0011 : 11 reg1 reg2
from memory	0110 0011 : mod reg r/m
BOUND – Check Array Against Bounds	0110 0010 : mod[A] reg r/m
BSF – Bit Scan Forward	
register1, register2	0000 1111 : 1011 1100 : 11 reg1 reg2
memory, register	0000 1111 : 1011 1100 : mod reg r/m
BSR – Bit Scan Reverse	
register1, register2	0000 1111 : 1011 1101 : 11 reg1 reg2
memory, register	0000 1111 : 1011 1101 : mod reg r/m
BSWAP – Byte Swap	0000 1111 : 1100 1 reg
BT – Bit Test	
register, immediate	0000 1111 : 1011 1010 : 11 100 reg: imm8 data
memory, immediate	0000 1111 : 1011 1010 : mod 100 r/m : imm8 data
register1, register2	0000 1111 : 1010 0011 : 11 reg2 reg1
memory, reg	0000 1111 : 1010 0011 : mod reg r/m
BTC – Bit Test and Complement	
register, immediate	0000 1111 : 1011 1010 : 11 111 reg: imm8 data
memory, immediate	0000 1111 : 1011 1010 : mod 111 r/m : imm8 data
register1, register2	0000 1111 : 1011 1011 : 11 reg2 reg1
memory, reg	0000 1111 : 1011 1011 : mod reg r/m
BTR – Bit Test and Reset	
register, immediate	0000 1111 : 1011 1010 : 11 110 reg: imm8 data
memory, immediate	0000 1111 : 1011 1010 : mod 110 r/m : imm8 data
register1, register2	0000 1111 : 1011 0011 : 11 reg2 reg1
memory, reg	0000 1111 : 1011 0011 : mod reg r/m
BTS – Bit Test and Set	
register, immediate	0000 1111 : 1011 1010 : 11 101 reg: imm8 data
memory, immediate	0000 1111 : 1011 1010 : mod 101 r/m : imm8 data
register1, register2	0000 1111 : 1010 1011 : 11 reg2 reg1
memory, reg	0000 1111 : 1010 1011 : mod reg r/m
CALL – Call Procedure (in same segment)	
direct	1110 1000 : full displacement
register indirect	1111 1111 : 11 010 reg
memory indirect	1111 1111 : mod 010 r/m
CALL – Call Procedure (in other segment)	
direct	1001 1010 : unsigned full offset, selector
indirect	1111 1111 : mod 011 r/m

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

Instruction and Format	Encoding
CBW – Convert Byte to Word	1001 1000
CDQ – Convert Doubleword to Qword	1001 1001
CLC – Clear Carry Flag	1111 1000
CLD – Clear Direction Flag	1111 1100
CLI – Clear Interrupt Flag	1111 1010
CLTS – Clear Task-Switched Flag in CR0	0000 1111 : 0000 0110
CMC – Complement Carry Flag	1111 0101
CMP – Compare Two Operands	
register1 with register2	0011 100w : 11 reg1 reg2
register2 with register1	0011 101w : 11 reg1 reg2
memory with register	0011 100w : mod reg r/m
register with memory	0011 101w : mod reg r/m
immediate with register	1000 00sw : 11 111 reg : immediate data
immediate with AL, AX, or EAX	0011 110w : immediate data
immediate with memory	1000 00sw : mod 111 r/m : immediate data
CMPS/CMPSB/CMPSW/CMPSD – Compare String Operands	1010 011w
CMPXCHG – Compare and Exchange	
register1, register2	0000 1111 : 1011 000w : 11 reg2 reg1
memory, register	0000 1111 : 1011 000w : mod reg r/m
CPUID – CPU Identification	0000 1111 : 1010 0010
CWD – Convert Word to Doubleword	1001 1001
CWDE – Convert Word to Doubleword	1001 1000
DAA – Decimal Adjust AL after Addition	0010 0111
DAS – Decimal Adjust AL after Subtraction	0010 1111
DEC – Decrement by 1	
register	1111 111w : 11 001 reg
register (alternate encoding)	0100 1 reg
memory	1111 111w : mod 001 r/m
DIV – Unsigned Divide	
AL, AX, or EAX by register	1111 011w : 11 110 reg
AL, AX, or EAX by memory	1111 011w : mod 110 r/m
HLT – Halt	1111 0100
IDIV – Signed Divide	
AL, AX, or EAX by register	1111 011w : 11 111 reg
AL, AX, or EAX by memory	1111 011w : mod 111 r/m

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

Instruction and Format	Encoding
IMUL – Signed Multiply	
AL, AX, or EAX with register	1111 011w : 11 101 reg
AL, AX, or EAX with memory	1111 011w : mod 101 reg
register1 with register2	0000 1111 : 1010 1111 : 11 : reg1 reg2
register with memory	0000 1111 : 1010 1111 : mod reg r/m
register1 with immediate to register2	0110 10s1 : 11 reg1 reg2 : immediate data
memory with immediate to register	0110 10s1 : mod reg r/m : immediate data
IN – Input From Port	
fixed port	1110 010w : port number
variable port	1110 110w
INC – Increment by 1	
reg	1111 111w : 11 000 reg
reg (alternate encoding)	0100 0 reg
memory	1111 111w : mod 000 r/m
INS – Input from DX Port	0110 110w
INT n – Interrupt Type n	1100 1101 : type
INT – Single-Step Interrupt 3	1100 1100
INTO – Interrupt 4 on Overflow	1100 1110
INVD – Invalidate Cache	0000 1111 : 0000 1000
INVLPG – Invalidate TLB Entry	0000 1111 : 0000 0001 : mod 111 r/m
INVPCID – Invalidate Process-Context Identifier	0110 0110:0000 1111:0011 1000:1000 0010: mod reg r/m
IRET/IRETD – Interrupt Return	1100 1111
***Jcc* – Jump if Condition is Met**	
8-bit displacement	0111 tttn : 8-bit displacement
full displacement	0000 1111 : 1000 tttn : full displacement
JCXZ/JECXZ – Jump on CX/ECX Zero Address-size prefix differentiates JCXZ and JECXZ	1110 0011 : 8-bit displacement
JMP – Unconditional Jump (to same segment)	
short	1110 1011 : 8-bit displacement
direct	1110 1001 : full displacement
register indirect	1111 1111 : 11 100 reg
memory indirect	1111 1111 : mod 100 r/m
JMP – Unconditional Jump (to other segment)	
direct intersegment	1110 1010 : unsigned full offset, selector
indirect intersegment	1111 1111 : mod 101 r/m
LAHF – Load Flags into AHRegister	1001 1111

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

Instruction and Format	Encoding
LAR – Load Access Rights Byte	
from register	0000 1111 : 0000 0010 : 11 reg1 reg2
from memory	0000 1111 : 0000 0010 : mod reg r/m
LDS – Load Pointer to DS	1100 0101 : mod[A,B] reg r/m
LEA – Load Effective Address	1000 1101 : mod[A] reg r/m
LEAVE – High Level Procedure Exit	1100 1001
LES – Load Pointer to ES	1100 0100 : mod[A,B] reg r/m
LFS – Load Pointer to FS	0000 1111 : 1011 0100 : mod[A] reg r/m
LGDT – Load Global Descriptor Table Register	0000 1111 : 0000 0001 : mod[A] 010 r/m
LGS – Load Pointer to GS	0000 1111 : 1011 0101 : mod[A] reg r/m
LIDT – Load Interrupt Descriptor Table Register	0000 1111 : 0000 0001 : mod[A] 011 r/m
LLDT – Load Local Descriptor Table Register	
LDTR from register	0000 1111 : 0000 0000 : 11 010 reg
LDTR from memory	0000 1111 : 0000 0000 : mod 010 r/m
LMSW – Load Machine Status Word	
from register	0000 1111 : 0000 0001 : 11 110 reg
from memory	0000 1111 : 0000 0001 : mod 110 r/m
LOCK – Assert LOCK# Signal Prefix	1111 0000
LODS/LODSB/LODSW/LODSD – Load String Operand	1010 110w
LOOP – Loop Count	1110 0010 : 8-bit displacement
LOOPZ/LOOPE – Loop Count while Zero/Equal	1110 0001 : 8-bit displacement
LOOPNZ/LOOPNE – Loop Count while not Zero/Equal	1110 0000 : 8-bit displacement
LSL – Load Segment Limit	
from register	0000 1111 : 0000 0011 : 11 reg1 reg2
from memory	0000 1111 : 0000 0011 : mod reg r/m
LSS – Load Pointer to SS	0000 1111 : 1011 0010 : mod[A] reg r/m
LTR – Load Task Register	
from register	0000 1111 : 0000 0000 : 11 011 reg
from memory	0000 1111 : 0000 0000 : mod 011 r/m
MOV – Move Data	
register1 to register2	1000 100w : 11 reg1 reg2
register2 to register1	1000 101w : 11 reg1 reg2
memory to reg	1000 101w : mod reg r/m
reg to memory	1000 100w : mod reg r/m
immediate to register	1100 011w : 11 000 reg : immediate data
immediate to register (alternate encoding)	1011 w reg : immediate data
immediate to memory	1100 011w : mod 000 r/m : immediate data
memory to AL, AX, or EAX	1010 000w : full displacement

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

Instruction and Format	Encoding
AL, AX, or EAX to memory	1010 001w : full displacement
MOV – Move to/from Control Registers	
CR0 from register	0000 1111 : 0010 0010 : -- 000 reg
CR2 from register	0000 1111 : 0010 0010 : -- 010reg
CR3 from register	0000 1111 : 0010 0010 : -- 011 reg
CR4 from register	0000 1111 : 0010 0010 : -- 100 reg
register from CR0-CR4	0000 1111 : 0010 0000 : -- eee reg
MOV – Move to/from Debug Registers	
DR0-DR3 from register	0000 1111 : 0010 0011 : -- eee reg
DR4-DR5 from register	0000 1111 : 0010 0011 : -- eee reg
DR6-DR7 from register	0000 1111 : 0010 0011 : -- eee reg
register from DR6-DR7	0000 1111 : 0010 0001 : -- eee reg
register from DR4-DR5	0000 1111 : 0010 0001 : -- eee reg
register from DR0-DR3	0000 1111 : 0010 0001 : -- eee reg
MOV – Move to/from Segment Registers	
register to segment register	1000 1110 : 11 sreg3 reg
register to SS	1000 1110 : 11 sreg3 reg
memory to segment reg	1000 1110 : mod sreg3 r/m
memory to SS	1000 1110 : mod sreg3 r/m
segment register to register	1000 1100 : 11 sreg3 reg
segment register to memory	1000 1100 : mod sreg3 r/m
MOVBE – Move data after swapping bytes	
memory to register	0000 1111 : 0011 1000:1111 0000 : mod reg r/m
register to memory	0000 1111 : 0011 1000:1111 0001 : mod reg r/m
MOVS/MOVSB/MOVSW/MOVSD – Move Data from String to String	1010 010w
MOVSX – Move with Sign-Extend	
memory to reg	0000 1111 : 1011 111w : mod reg r/m
MOVZX – Move with Zero-Extend	
register2 to register1	0000 1111 : 1011 011w : 11 reg1 reg2
memory to register	0000 1111 : 1011 011w : mod reg r/m
MUL – Unsigned Multiply	
AL, AX, or EAX with register	1111 011w : 11 100 reg
AL, AX, or EAX with memory	1111 011w : mod 100 r/m
NEG – Two's Complement Negation	
register	1111 011w : 11 011 reg
memory	1111 011w : mod 011 r/m
NOP – No Operation	1001 0000

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

Instruction and Format	Encoding
NOP – Multi-byte No Operation[1]	
register	0000 1111 0001 1111 : 11 000 reg
memory	0000 1111 0001 1111 : mod 000 r/m
NOT – One's Complement Negation	
register	1111 011w : 11 010 reg
memory	1111 011w : mod 010 r/m
OR – Logical Inclusive OR	
register1 to register2	0000 100w : 11 reg1 reg2
register2 to register1	0000 101w : 11 reg1 reg2
memory to register	0000 101w : mod reg r/m
register to memory	0000 100w : mod reg r/m
immediate to register	1000 00sw : 11 001 reg : immediate data
immediate to AL, AX, or EAX	0000 110w : immediate data
immediate to memory	1000 00sw : mod 001 r/m : immediate data
OUT – Output to Port	
fixed port	1110 011w : port number
variable port	1110 111w
OUTS – Output to DX Port	0110 111w
POP – Pop a Word from the Stack	
register	1000 1111 : 11 000 reg
register (alternate encoding)	0101 1 reg
memory	1000 1111 : mod 000 r/m
POP – Pop a Segment Register from the Stack (Note: CS cannot be sreg2 in this usage.)	
segment register DS, ES	000 sreg2 111
segment register SS	000 sreg2 111
segment register FS, GS	0000 1111: 10 sreg3 001
POPA/POPAD – Pop All General Registers	0110 0001
POPF/POPFD – Pop Stack into FLAGS or EFLAGS Register	1001 1101
PUSH – Push Operand onto the Stack	
register	1111 1111 : 11 110 reg
register (alternate encoding)	0101 0 reg
memory	1111 1111 : mod 110 r/m
immediate	0110 10s0 : immediate data
PUSH – Push Segment Register onto the Stack	
segment register CS,DS,ES,SS	000 sreg2 110
segment register FS,GS	0000 1111: 10 sreg3 000
PUSHA/PUSHAD – Push All General Registers	0110 0000

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

Instruction and Format	Encoding
PUSHF/PUSHFD – Push Flags Register onto the Stack	1001 1100
RCL – Rotate thru Carry Left	
register by 1	1101 000w : 11 010 reg
memory by 1	1101 000w : mod 010 r/m
register by CL	1101 001w : 11 010 reg
memory by CL	1101 001w : mod 010 r/m
register by immediate count	1100 000w : 11 010 reg : imm8 data
memory by immediate count	1100 000w : mod 010 r/m : imm8 data
RCR – Rotate thru Carry Right	
register by 1	1101 000w : 11 011 reg
memory by 1	1101 000w : mod 011 r/m
register by CL	1101 001w : 11 011 reg
memory by CL	1101 001w : mod 011 r/m
register by immediate count	1100 000w : 11 011 reg : imm8 data
memory by immediate count	1100 000w : mod 011 r/m : imm8 data
RDMSR – Read from Model-Specific Register	0000 1111 : 0011 0010
RDPMC – Read Performance Monitoring Counters	0000 1111 : 0011 0011
RDTSC – Read Time-Stamp Counter	0000 1111 : 0011 0001
RDTSCP – Read Time-Stamp Counter and Processor ID	0000 1111 : 0000 0001: 1111 1001
REP INS – Input String	1111 0011 : 0110 110w
REP LODS – Load String	1111 0011 : 1010 110w
REP MOVS – Move String	1111 0011 : 1010 010w
REP OUTS – Output String	1111 0011 : 0110 111w
REP STOS – Store String	1111 0011 : 1010 101w
REPE CMPS – Compare String	1111 0011 : 1010 011w
REPE SCAS – Scan String	1111 0011 : 1010 111w
REPNE CMPS – Compare String	1111 0010 : 1010 011w
REPNE SCAS – Scan String	1111 0010 : 1010 111w
RET – Return from Procedure (to same segment)	
no argument	1100 0011
adding immediate to SP	1100 0010 : 16-bit displacement
RET – Return from Procedure (to other segment)	
intersegment	1100 1011
adding immediate to SP	1100 1010 : 16-bit displacement

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

Instruction and Format	Encoding
ROL – Rotate Left	
register by 1	1101 000w : 11 000 reg
memory by 1	1101 000w : mod 000 r/m
register by CL	1101 001w : 11 000 reg
memory by CL	1101 001w : mod 000 r/m
register by immediate count	1100 000w : 11 000 reg : imm8 data
memory by immediate count	1100 000w : mod 000 r/m : imm8 data
ROR – Rotate Right	
register by 1	1101 000w : 11 001 reg
memory by 1	1101 000w : mod 001 r/m
register by CL	1101 001w : 11 001 reg
memory by CL	1101 001w : mod 001 r/m
register by immediate count	1100 000w : 11 001 reg : imm8 data
memory by immediate count	1100 000w : mod 001 r/m : imm8 data
RSM – Resume from System Management Mode	0000 1111 : 1010 1010
SAHF – Store AH into Flags	1001 1110
SAL – Shift Arithmetic Left	same instruction as SHL
SAR – Shift Arithmetic Right	
register by 1	1101 000w : 11 111 reg
memory by 1	1101 000w : mod 111 r/m
register by CL	1101 001w : 11 111 reg
memory by CL	1101 001w : mod 111 r/m
register by immediate count	1100 000w : 11 111 reg : imm8 data
memory by immediate count	1100 000w : mod 111 r/m : imm8 data
SBB – Integer Subtraction with Borrow	
register1 to register2	0001 100w : 11 reg1 reg2
register2 to register1	0001 101w : 11 reg1 reg2
memory to register	0001 101w : mod reg r/m
register to memory	0001 100w : mod reg r/m
immediate to register	1000 00sw : 11 011 reg : immediate data
immediate to AL, AX, or EAX	0001 110w : immediate data
immediate to memory	1000 00sw : mod 011 r/m : immediate data
SCAS/SCASB/SCASW/SCASD – Scan String	1010 111w
SETcc – Byte Set on Condition	
register	0000 1111 : 1001 tttn : 11 000 reg
memory	0000 1111 : 1001 tttn : mod 000 r/m
SGDT – Store Global Descriptor Table Register	0000 1111 : 0000 0001 : mod[A] 000 r/m

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

Instruction and Format	Encoding
SHL – Shift Left	
register by 1	1101 000w : 11 100 reg
memory by 1	1101 000w : mod 100 r/m
register by CL	1101 001w : 11 100 reg
memory by CL	1101 001w : mod 100 r/m
register by immediate count	1100 000w : 11 100 reg : imm8 data
memory by immediate count	1100 000w : mod 100 r/m : imm8 data
SHLD – Double Precision Shift Left	
register by immediate count	0000 1111 : 1010 0100 : 11 reg2 reg1 : imm8
memory by immediate count	0000 1111 : 1010 0100 : mod reg r/m : imm8
register by CL	0000 1111 : 1010 0101 : 11 reg2 reg1
memory by CL	0000 1111 : 1010 0101 : mod reg r/m
SHR – Shift Right	
register by 1	1101 000w : 11 101 reg
memory by 1	1101 000w : mod 101 r/m
register by CL	1101 001w : 11 101 reg
memory by CL	1101 001w : mod 101 r/m
register by immediate count	1100 000w : 11 101 reg : imm8 data
memory by immediate count	1100 000w : mod 101 r/m : imm8 data
SHRD – Double Precision Shift Right	
register by immediate count	0000 1111 : 1010 1100 : 11 reg2 reg1 : imm8
memory by immediate count	0000 1111 : 1010 1100 : mod reg r/m : imm8
register by CL	0000 1111 : 1010 1101 : 11 reg2 reg1
memory by CL	0000 1111 : 1010 1101 : mod reg r/m
SIDT – Store Interrupt Descriptor Table Register	0000 1111 : 0000 0001 : mod[A] 001 r/m
SLDT – Store Local Descriptor Table Register	
to register	0000 1111 : 0000 0000 : 11 000 reg
to memory	0000 1111 : 0000 0000 : mod 000 r/m
SMSW – Store Machine Status Word	
to register	0000 1111 : 0000 0001 : 11 100 reg
to memory	0000 1111 : 0000 0001 : mod 100 r/m
STC – Set Carry Flag	1111 1001
STD – Set Direction Flag	1111 1101
STI – Set Interrupt Flag	1111 1011
STOS/STOSB/STOSW/STOSD – Store String Data	1010 101w
STR – Store Task Register	
to register	0000 1111 : 0000 0000 : 11 001 reg
to memory	0000 1111 : 0000 0000 : mod 001 r/m

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

Instruction and Format	Encoding
SUB – Integer Subtraction	
register1 to register2	0010 100w : 11 reg1 reg2
register2 to register1	0010 101w : 11 reg1 reg2
memory to register	0010 101w : mod reg r/m
register to memory	0010 100w : mod reg r/m
immediate to register	1000 00sw : 11 101 reg : immediate data
immediate to AL, AX, or EAX	0010 110w : immediate data
immediate to memory	1000 00sw : mod 101 r/m : immediate data
TEST – Logical Compare	
register1 and register2	1000 010w : 11 reg1 reg2
memory and register	1000 010w : mod reg r/m
immediate and register	1111 011w : 11 000 reg : immediate data
immediate and AL, AX, or EAX	1010 100w : immediate data
immediate and memory	1111 011w : mod 000 r/m : immediate data
UD0 – Undefined instruction	0000 1111 : 1111 1111
UD1 – Undefined instruction	0000 1111 : 0000 1011
UD2 – Undefined instruction	0000 FFFF : 0000 1011
VERR – Verify a Segment for Reading	
register	0000 1111 : 0000 0000 : 11 100 reg
memory	0000 1111 : 0000 0000 : mod 100 r/m
VERW – Verify a Segment for Writing	
register	0000 1111 : 0000 0000 : 11 101 reg
memory	0000 1111 : 0000 0000 : mod 101 r/m
WAIT – Wait	1001 1011
WBINVD – Writeback and Invalidate Data Cache	0000 1111 : 0000 1001
WRMSR – Write to Model-Specific Register	0000 1111 : 0011 0000
XADD – Exchange and Add	
register1, register2	0000 1111 : 1100 000w : 11 reg2 reg1
memory, reg	0000 1111 : 1100 000w : mod reg r/m
XCHG – Exchange Register/Memory with Register	
register1 with register2	1000 011w : 11 reg1 reg2
AX or EAX with reg	1001 0 reg
memory with reg	1000 011w : mod reg r/m
XLAT/XLATB – Table Look-up Translation	1101 0111
XOR – Logical Exclusive OR	
register1 to register2	0011 000w : 11 reg1 reg2
register2 to register1	0011 001w : 11 reg1 reg2
memory to register	0011 001w : mod reg r/m

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

Instruction and Format	Encoding
register to memory	0011 000w : mod reg r/m
immediate to register	1000 00sw : 11 110 reg : immediate data
immediate to AL, AX, or EAX	0011 010w : immediate data
immediate to memory	1000 00sw : mod 110 r/m : immediate data
Prefix Bytes	
address size	0110 0111
LOCK	1111 0000
operand size	0110 0110
CS segment override	0010 1110
DS segment override	0011 1110
ES segment override	0010 0110
FS segment override	0110 0100
GS segment override	0110 0101
SS segment override	0011 0110

NOTES:

1. The multi-byte NOP instruction does not alter the content of the register and will not issue a memory operation.

B.2.1 General Purpose Instruction Formats and Encodings for 64-Bit Mode

Table B-15 shows machine instruction formats and encodings for general purpose instructions in 64-bit mode.

Table B-14. Special Symbols

Symbol	Application
S	If the value of REX.W. is 1, it overrides the presence of 66H.
w	The value of bit W. in REX is has no effect.

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode

Instruction and Format	Encoding
ADC – ADD with Carry	
register1 to register2	0100 0R0B : 0001 000w : 11 reg1 reg2
qwordregister1 to qwordregister2	0100 1R0B : 0001 0001 : 11 qwordreg1 qwordreg2
register2 to register1	0100 0R0B : 0001 001w : 11 reg1 reg2
qwordregister1 to qwordregister2	0100 1R0B : 0001 0011 : 11 qwordreg1 qwordreg2
memory to register	0100 0RXB : 0001 001w : mod reg r/m
memory to qwordregister	0100 1RXB : 0001 0011 : mod qwordreg r/m
register to memory	0100 0RXB : 0001 000w : mod reg r/m
qwordregister to memory	0100 1RXB : 0001 0001 : mod qwordreg r/m
immediate to register	0100 000B : 1000 00sw : 11 010 reg : immediate
immediate to qwordregister	0100 100B : 1000 0001 : 11 010 qwordreg : imm32
immediate to qwordregister	0100 1R0B : 1000 0011 : 11 010 qwordreg : imm8

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
immediate to AL, AX, or EAX	0001 010w : immediate data
immediate to RAX	0100 1000 : 0000 0101 : imm32
immediate to memory	0100 00XB : 1000 00sw : mod 010 r/m : immediate
immediate32 to memory64	0100 10XB : 1000 0001 : mod 010 r/m : imm32
immediate8 to memory64	0100 10XB : 1000 0031 : mod 010 r/m : imm8
ADD – Add	
register1 to register2	0100 0R0B : 0000 000w : 11 reg1 reg2
qwordregister1 to qwordregister2	0100 1R0B 0000 0000 : 11 qwordreg1 qwordreg2
register2 to register1	0100 0R0B : 0000 001w : 11 reg1 reg2
qwordregister1 to qwordregister2	0100 1R0B 0000 0010 : 11 qwordreg1 qwordreg2
memory to register	0100 0RXB : 0000 001w : mod reg r/m
memory64 to qwordregister	0100 1RXB : 0000 0000 : mod qwordreg r/m
register to memory	0100 0RXB : 0000 000w : mod reg r/m
qwordregister to memory64	0100 1RXB : 0000 0011 : mod qwordreg r/m
immediate to register	0100 0000B : 1000 00sw : 11 000 reg : immediate data
immediate32 to qwordregister	0100 100B : 1000 0001 : 11 010 qwordreg : imm
immediate to AL, AX, or EAX	0000 010w : immediate8
immediate to RAX	0100 1000 : 0000 0101 : imm32
immediate to memory	0100 00XB : 1000 00sw : mod 000 r/m : immediate
immediate32 to memory64	0100 10XB : 1000 0001 : mod 010 r/m : imm32
immediate8 to memory64	0100 10XB : 1000 0011 : mod 010 r/m : imm8
AND – Logical AND	
register1 to register2	0100 0R0B 0010 000w : 11 reg1 reg2
qwordregister1 to qwordregister2	0100 1R0B 0010 0001 : 11 qwordreg1 qwordreg2
register2 to register1	0100 0R0B 0010 001w : 11 reg1 reg2
register1 to register2	0100 1R0B 0010 0011 : 11 qwordreg1 qwordreg2
memory to register	0100 0RXB 0010 001w : mod reg r/m
memory64 to qwordregister	0100 1RXB : 0010 0011 : mod qwordreg r/m
register to memory	0100 0RXB : 0010 000w : mod reg r/m
qwordregister to memory64	0100 1RXB : 0010 0001 : mod qwordreg r/m
immediate to register	0100 000B : 1000 00sw : 11 100 reg : immediate
immediate32 to qwordregister	0100 100B 1000 0001 : 11 100 qwordreg : imm32
immediate to AL, AX, or EAX	0010 010w : immediate
immediate32 to RAX	0100 1000 0010 1001 : imm32
immediate to memory	0100 00XB : 1000 00sw : mod 100 r/m : immediate
immediate32 to memory64	0100 10XB : 1000 0001 : mod 100 r/m : immediate32
immediate8 to memory64	0100 10XB : 1000 0011 : mod 100 r/m : imm8
BSF – Bit Scan Forward	

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
register1, register2	0100 0R0B 0000 1111 : 1011 1100 : 11 reg1 reg2
qwordregister1, qwordregister2	0100 1R0B 0000 1111 : 1011 1100 : 11 qwordreg1 qwordreg2
memory, register	0100 0RXB 0000 1111 : 1011 1100 : mod reg r/m
memory64, qwordregister	0100 1RXB 0000 1111 : 1011 1100 : mod qwordreg r/m
BSR – Bit Scan Reverse	
register1, register2	0100 0R0B 0000 1111 : 1011 1101 : 11 reg1 reg2
qwordregister1, qwordregister2	0100 1R0B 0000 1111 : 1011 1101 : 11 qwordreg1 qwordreg2
memory, register	0100 0RXB 0000 1111 : 1011 1101 : mod reg r/m
memory64, qwordregister	0100 1RXB 0000 1111 : 1011 1101 : mod qwordreg r/m
BSWAP – Byte Swap	0000 1111 : 1100 1 reg
BSWAP – Byte Swap	0100 100B 0000 1111 : 1100 1 qwordreg
BT – Bit Test	
register, immediate	0100 000B 0000 1111 : 1011 1010 : 11 100 reg: imm8
qwordregister, immediate8	0100 100B 1111 : 1011 1010 : 11 100 qwordreg: imm8 data
memory, immediate	0100 00XB 0000 1111 : 1011 1010 : mod 100 r/m : imm8
memory64, immediate8	0100 10XB 0000 1111 : 1011 1010 : mod 100 r/m : imm8 data
register1, register2	0100 0R0B 0000 1111 : 1010 0011 : 11 reg2 reg1
qwordregister1, qwordregister2	0100 1R0B 0000 1111 : 1010 0011 : 11 qwordreg2 qwordreg1
memory, reg	0100 0RXB 0000 1111 : 1010 0011 : mod reg r/m
memory, qwordreg	0100 1RXB 0000 1111 : 1010 0011 : mod qwordreg r/m
BTC – Bit Test and Complement	
register, immediate	0100 000B 0000 1111 : 1011 1010 : 11 111 reg: imm8
qwordregister, immediate8	0100 100B 0000 1111 : 1011 1010 : 11 111 qwordreg: imm8
memory, immediate	0100 00XB 0000 1111 : 1011 1010 : mod 111 r/m : imm8
memory64, immediate8	0100 10XB 0000 1111 : 1011 1010 : mod 111 r/m : imm8
register1, register2	0100 0R0B 0000 1111 : 1011 1011 : 11 reg2 reg1
qwordregister1, qwordregister2	0100 1R0B 0000 1111 : 1011 1011 : 11 qwordreg2 qwordreg1
memory, register	0100 0RXB 0000 1111 : 1011 1011 : mod reg r/m
memory, qwordreg	0100 1RXB 0000 1111 : 1011 1011 : mod qwordreg r/m
BTR – Bit Test and Reset	
register, immediate	0100 000B 0000 1111 : 1011 1010 : 11 110 reg: imm8
qwordregister, immediate8	0100 100B 0000 1111 : 1011 1010 : 11 110 qwordreg: imm8
memory, immediate	0100 00XB 0000 1111 : 1011 1010 : mod 110 r/m : imm8
memory64, immediate8	0100 10XB 0000 1111 : 1011 1010 : mod 110 r/m : imm8
register1, register2	0100 0R0B 0000 1111 : 1011 0011 : 11 reg2 reg1

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
qwordregister1, qwordregister2	0100 1R0B 0000 1111 : 1011 0011 : 11 qwordreg2 qwordreg1
memory, register	0100 0RXB 0000 1111 : 1011 0011 : mod reg r/m
memory64, qwordreg	0100 1RXB 0000 1111 : 1011 0011 : mod qwordreg r/m
BTS – Bit Test and Set	
register, immediate	0100 000B 0000 1111 : 1011 1010 : 11 101 reg: imm8
qwordregister, immediate8	0100 100B 0000 1111 : 1011 1010 : 11 101 qwordreg: imm8
memory, immediate	0100 00XB 0000 1111 : 1011 1010 : mod 101 r/m : imm8
memory64, immediate8	0100 10XB 0000 1111 : 1011 1010 : mod 101 r/m : imm8
register1, register2	0100 0R0B 0000 1111 : 1010 1011 : 11 reg2 reg1
qwordregister1, qwordregister2	0100 1R0B 0000 1111 : 1010 1011 : 11 qwordreg2 qwordreg1
memory, register	0100 0RXB 0000 1111 : 1010 1011 : mod reg r/m
memory64, qwordreg	0100 1RXB 0000 1111 : 1010 1011 : mod qwordreg r/m
CALL – Call Procedure (in same segment)	
direct	1110 1000 : displacement32
register indirect	0100 WR00^{w} 1111 1111 : 11 010 reg
memory indirect	0100 W0XB^{w} 1111 1111 : mod 010 r/m
CALL – Call Procedure (in other segment)	
indirect	1111 1111 : mod 011 r/m
indirect	0100 10XB 0100 1000 1111 1111 : mod 011 r/m
CBW – Convert Byte to Word	1001 1000
CDQ – Convert Doubleword to Qword+	1001 1001
CDQE – RAX, Sign-Extend of EAX	0100 1000 1001 1001
CLC – Clear Carry Flag	1111 1000
CLD – Clear Direction Flag	1111 1100
CLI – Clear Interrupt Flag	1111 1010
CLTS – Clear Task-Switched Flag in CR0	0000 1111 : 0000 0110
CMC – Complement Carry Flag	1111 0101
CMP – Compare Two Operands	
register1 with register2	0100 0R0B 0011 100w : 11 reg1 reg2
qwordregister1 with qwordregister2	0100 1R0B 0011 1001 : 11 qwordreg1 qwordreg2
register2 with register1	0100 0R0B 0011 101w : 11 reg1 reg2
qwordregister2 with qwordregister1	0100 1R0B 0011 101w : 11 qwordreg1 qwordreg2
memory with register	0100 0RXB 0011 100w : mod reg r/m
memory64 with qwordregister	0100 1RXB 0011 1001 : mod qwordreg r/m
register with memory	0100 0RXB 0011 101w : mod reg r/m
qwordregister with memory64	0100 1RXB 0011 101w1 : mod qwordreg r/m
immediate with register	0100 000B 1000 00sw : 11 111 reg : imm

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
immediate32 with qwordregister	0100 100B 1000 0001 : 11 111 qwordreg : imm64
immediate with AL, AX, or EAX	0011 110w : imm
immediate32 with RAX	0100 1000 0011 1101 : imm32
immediate with memory	0100 00XB 1000 00sw : mod 111 r/m : imm
immediate32 with memory64	0100 1RXB 1000 0001 : mod 111 r/m : imm64
immediate8 with memory64	0100 1RXB 1000 0011 : mod 111 r/m : imm8
CMPS/CMPSB/CMPSW/CMPSD/CMPSQ – Compare String Operands	
compare string operands [X at DS:(E)SI with Y at ES:(E)DI]	1010 011w
qword at address RSI with qword at address RDI	0100 1000 1010 0111
CMPXCHG – Compare and Exchange	
register1, register2	0000 1111 : 1011 000w : 11 reg2 reg1
byteregister1, byteregister2	0100 000B 0000 1111 : 1011 0000 : 11 bytereg2 reg1
qwordregister1, qwordregister2	0100 100B 0000 1111 : 1011 0001 : 11 qwordreg2 reg1
memory, register	0000 1111 : 1011 000w : mod reg r/m
memory8, byteregister	0100 00XB 0000 1111 : 1011 0000 : mod bytereg r/m
memory64, qwordregister	0100 10XB 0000 1111 : 1011 0001 : mod qwordreg r/m
CPUID – CPU Identification	0000 1111 : 1010 0010
CQO – Sign-Extend RAX	0100 1000 1001 1001
CWD – Convert Word to Doubleword	1001 1001
CWDE – Convert Word to Doubleword	1001 1000
DEC – Decrement by 1	
register	0100 000B 1111 111w : 11 001 reg
qwordregister	0100 100B 1111 1111 : 11 001 qwordreg
memory	0100 00XB 1111 111w : mod 001 r/m
memory64	0100 10XB 1111 1111 : mod 001 r/m
DIV – Unsigned Divide	
AL, AX, or EAX by register	0100 000B 1111 011w : 11 110 reg
Divide RDX:RAX by qwordregister	0100 100B 1111 0111 : 11 110 qwordreg
AL, AX, or EAX by memory	0100 00XB 1111 011w : mod 110 r/m
Divide RDX:RAX by memory64	0100 10XB 1111 0111 : mod 110 r/m
ENTER – Make Stack Frame for High Level Procedure	1100 1000 : 16-bit displacement : 8-bit level (L)
HLT – Halt	1111 0100
IDIV – Signed Divide	
AL, AX, or EAX by register	0100 000B 1111 011w : 11 111 reg
RDX:RAX by qwordregister	0100 100B 1111 0111 : 11 111 qwordreg
AL, AX, or EAX by memory	0100 00XB 1111 011w : mod 111 r/m
RDX:RAX by memory64	0100 10XB 1111 0111 : mod 111 r/m
IMUL – Signed Multiply	

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
AL, AX, or EAX with register	0100 000B 1111 011w : 11 101 reg
RDX:RAX := RAX with qwordregister	0100 100B 1111 0111 : 11 101 qwordreg
AL, AX, or EAX with memory	0100 00XB 1111 011w : mod 101 r/m
RDX:RAX := RAX with memory64	0100 10XB 1111 0111 : mod 101 r/m
register1 with register2	0000 1111 : 1010 1111 : 11 : reg1 reg2
qwordregister1 := qwordregister1 with qwordregister2	0100 1R0B 0000 1111 : 1010 1111 : 11 : qwordreg1 qwordreg2
register with memory	0100 0RXB 0000 1111 : 1010 1111 : mod reg r/m
qwordregister := qwordregister with memory64	0100 1RXB 0000 1111 : 1010 1111 : mod qwordreg r/m
register1 with immediate to register2	0100 0R0B 0110 10s1 : 11 reg1 reg2 : imm
qwordregister1 := qwordregister2 with sign-extended immediate8	0100 1R0B 0110 1011 : 11 qwordreg1 qwordreg2 : imm8
qwordregister1 := qwordregister2 with immediate32	0100 1R0B 0110 1001 : 11 qwordreg1 qwordreg2 : imm32
memory with immediate to register	0100 0RXB 0110 10s1 : mod reg r/m : imm
qwordregister := memory64 with sign-extended immediate8	0100 1RXB 0110 1011 : mod qwordreg r/m : imm8
qwordregister := memory64 with immediate32	0100 1RXB 0110 1001 : mod qwordreg r/m : imm32
IN – Input From Port	
fixed port	1110 010w : port number
variable port	1110 110w
INC – Increment by 1	
reg	0100 000B 1111 111w : 11 000 reg
qwordreg	0100 100B 1111 1111 : 11 000 qwordreg
memory	0100 00XB 1111 111w : mod 000 r/m
memory64	0100 10XB 1111 1111 : mod 000 r/m
INS – Input from DX Port	0110 110w
INT n – Interrupt Type n	1100 1101 : type
INT – Single-Step Interrupt 3	1100 1100
INTO – Interrupt 4 on Overflow	1100 1110
INVD – Invalidate Cache	0000 1111 : 0000 1000
INVLPG – Invalidate TLB Entry	0000 1111 : 0000 0001 : mod 111 r/m
INVPCID – Invalidate Process-Context Identifier	0110 0110:0000 1111:0011 1000:1000 0010: mod reg r/m
IRETO – Interrupt Return	1100 1111
***Jcc* – Jump if Condition is Met**	
8-bit displacement	0111 tttn : 8-bit displacement
displacements (excluding 16-bit relative offsets)	0000 1111 : 1000 tttn : displacement32
JCXZ/JECXZ – Jump on CX/ECX Zero	
Address-size prefix differentiates JCXZ and JECXZ	1110 0011 : 8-bit displacement
JMP – Unconditional Jump (to same segment)	
short	1110 1011 : 8-bit displacement

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
direct	1110 1001 : displacement32
register indirect	0100 W00B[W] : 1111 1111 : 11 100 reg
memory indirect	0100 W0XB[W] : 1111 1111 : mod 100 r/m
JMP – Unconditional Jump (to other segment)	
indirect intersegment	0100 00XB : 1111 1111 : mod 101 r/m
64-bit indirect intersegment	0100 10XB : 1111 1111 : mod 101 r/m
LAR – Load Access Rights Byte	
from register	0100 0R0B : 0000 1111 : 0000 0010 : 11 reg1 reg2
from dwordregister to qwordregister, masked by 00FxFF00H	0100 WR0B : 0000 1111 : 0000 0010 : 11 qwordreg1 dwordreg2
from memory	0100 0RXB : 0000 1111 : 0000 0010 : mod reg r/m
from memory32 to qwordregister, masked by 00FxFF00H	0100 WRXB 0000 1111 : 0000 0010 : mod r/m
LEA – Load Effective Address	
in wordregister/dwordregister	0100 0RXB : 1000 1101 : mod[A] reg r/m
in qwordregister	0100 1RXB : 1000 1101 : mod[A] qwordreg r/m
LEAVE – High Level Procedure Exit	1100 1001
LFS – Load Pointer to FS	
FS:r16/r32 with far pointer from memory	0100 0RXB : 0000 1111 : 1011 0100 : mod[A] reg r/m
FS:r64 with far pointer from memory	0100 1RXB : 0000 1111 : 1011 0100 : mod[A] qwordreg r/m
LGDT – Load Global Descriptor Table Register	0100 10XB : 0000 1111 : 0000 0001 : mod[A] 010 r/m
LGS – Load Pointer to GS	
GS:r16/r32 with far pointer from memory	0100 0RXB : 0000 1111 : 1011 0101 : mod[A] reg r/m
GS:r64 with far pointer from memory	0100 1RXB : 0000 1111 : 1011 0101 : mod[A] qwordreg r/m
LIDT – Load Interrupt Descriptor Table Register	0100 10XB : 0000 1111 : 0000 0001 : mod[A] 011 r/m
LLDT – Load Local Descriptor Table Register	
LDTR from register	0100 000B : 0000 1111 : 0000 0000 : 11 010 reg
LDTR from memory	0100 00XB :0000 1111 : 0000 0000 : mod 010 r/m
LMSW – Load Machine Status Word	
from register	0100 000B : 0000 1111 : 0000 0001 : 11 110 reg
from memory	0100 00XB :0000 1111 : 0000 0001 : mod 110 r/m
LOCK – Assert LOCK# Signal Prefix	1111 0000
LODS/LODSB/LODSW/LODSD/LODSQ – Load String Operand	
at DS:(E)SI to AL/EAX/EAX	1010 110w
at (R)SI to RAX	0100 1000 1010 1101
LOOP – Loop Count	
if count ≠ 0, 8-bit displacement	1110 0010
if count ≠ 0, RIP + 8-bit displacement sign-extended to 64-bits	0100 1000 1110 0010
LOOPE – Loop Count while Zero/Equal	

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
if count ≠ 0 & ZF =1, 8-bit displacement	1110 0001
if count ≠ 0 & ZF = 1, RIP + 8-bit displacement sign-extended to 64-bits	0100 1000 1110 0001
LOOPNE/LOOPNZ – Loop Count while not Zero/Equal	
if count ≠ 0 & ZF = 0, 8-bit displacement	1110 0000
if count ≠ 0 & ZF = 0, RIP + 8-bit displacement sign-extended to 64-bits	0100 1000 1110 0000
LSL – Load Segment Limit	
from register	0000 1111 : 0000 0011 : 11 reg1 reg2
from qwordregister	0100 1R00 0000 1111 : 0000 0011 : 11 qwordreg1 reg2
from memory16	0000 1111 : 0000 0011 : mod reg r/m
from memory64	0100 1RXB 0000 1111 : 0000 0011 : mod qwordreg r/m
LSS – Load Pointer to SS	
SS:r16/r32 with far pointer from memory	0100 0RXB : 0000 1111 : 1011 0010 : mod^{A} reg r/m
SS:r64 with far pointer from memory	0100 1WXB : 0000 1111 : 1011 0010 : mod^{A} qwordreg r/m
LTR – Load Task Register	
from register	0100 0R00 : 0000 1111 : 0000 0000 : 11 011 reg
from memory	0100 00XB : 0000 1111 : 0000 0000 : mod 011 r/m
MOV – Move Data	
register1 to register2	0100 0R0B : 1000 100w : 11 reg1 reg2
qwordregister1 to qwordregister2	0100 1R0B 1000 1001 : 11 qwordeg1 qwordreg2
register2 to register1	0100 0R0B : 1000 101w : 11 reg1 reg2
qwordregister2 to qwordregister1	0100 1R0B 1000 1011 : 11 qwordreg1 qwordreg2
memory to reg	0100 0RXB : 1000 101w : mod reg r/m
memory64 to qwordregister	0100 1RXB 1000 1011 : mod qwordreg r/m
reg to memory	0100 0RXB : 1000 100w : mod reg r/m
qwordregister to memory64	0100 1RXB 1000 1001 : mod qwordreg r/m
immediate to register	0100 000B : 1100 011w : 11 000 reg : imm
immediate32 to qwordregister (zero extend)	0100 100B 1100 0111 : 11 000 qwordreg : imm32
immediate to register (alternate encoding)	0100 000B : 1011 w reg : imm
immediate64 to qwordregister (alternate encoding)	0100 100B 1011 1000 reg : imm64
immediate to memory	0100 00XB : 1100 011w : mod 000 r/m : imm
immediate32 to memory64 (zero extend)	0100 10XB 1100 0111 : mod 000 r/m : imm32
memory to AL, AX, or EAX	0100 0000 : 1010 000w : displacement
memory64 to RAX	0100 1000 1010 0001 : displacement64
AL, AX, or EAX to memory	0100 0000 : 1010 001w : displacement
RAX to memory64	0100 1000 1010 0011 : displacement64
MOV – Move to/from Control Registers	
CR0-CR4 from register	0100 0R0B : 0000 1111 : 0010 0010 : 11 eee reg (eee = CR#)

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
CRx from qwordregister	0100 1R0B : 0000 1111 : 0010 0010 : 11 eee qwordreg (Reee = CR#)
register from CR0-CR4	0100 0R0B : 0000 1111 : 0010 0000 : 11 eee reg (eee = CR#)
qwordregister from CRx	0100 1R0B 0000 1111 : 0010 0000 : 11 eee qwordreg (Reee = CR#)
MOV – Move to/from Debug Registers	
DR0-DR7 from register	0000 1111 : 0010 0011 : 11 eee reg (eee = DR#)
DR0-DR7 from quadregister	0100 10OB 0000 1111 : 0010 0011 : 11 eee reg (eee = DR#)
register from DR0-DR7	0000 1111 : 0010 0001 : 11 eee reg (eee = DR#)
quadregister from DR0-DR7	0100 10OB 0000 1111 : 0010 0001 : 11 eee quadreg (eee = DR#)
MOV – Move to/from Segment Registers	
register to segment register	0100 W00B[W] : 1000 1110 : 11 sreg reg
register to SS	0100 000B : 1000 1110 : 11 sreg reg
memory to segment register	0100 00XB : 1000 1110 : mod sreg r/m
memory64 to segment register (lower 16 bits)	0100 10XB 1000 1110 : mod sreg r/m
memory to SS	0100 00XB : 1000 1110 : mod sreg r/m
segment register to register	0100 000B : 1000 1100 : 11 sreg reg
segment register to qwordregister (zero extended)	0100 100B 1000 1100 : 11 sreg qwordreg
segment register to memory	0100 00XB : 1000 1100 : mod sreg r/m
segment register to memory64 (zero extended)	0100 10XB 1000 1100 : mod sreg3 r/m
MOVBE – Move data after swapping bytes	
memory to register	0100 0RXB : 0000 1111 : 0011 1000:1111 0000 : mod reg r/m
memory64 to qwordregister	0100 1RXB : 0000 1111 : 0011 1000:1111 0000 : mod reg r/m
register to memory	0100 0RXB :0000 1111 : 0011 1000:1111 0001 : mod reg r/m
qwordregister to memory64	0100 1RXB :0000 1111 : 0011 1000:1111 0001 : mod reg r/m
MOVS/MOVSB/MOVSW/MOVSD/MOVSQ – Move Data from String to String	
Move data from string to string	1010 010w
Move data from string to string (qword)	0100 1000 1010 0101
MOVSX/MOVSXD – Move with Sign-Extend	
register2 to register1	0100 0R0B : 0000 1111 : 1011 111w : 11 reg1 reg2
byteregister2 to qwordregister1 (sign-extend)	0100 1R0B 0000 1111 : 1011 1110 : 11 quadreg1 bytereg2
wordregister2 to qwordregister1	0100 1R0B 0000 1111 : 1011 1111 : 11 quadreg1 wordreg2
dwordregister2 to qwordregister1	0100 1R0B 0110 0011 : 11 quadreg1 dwordreg2
memory to register	0100 0RXB : 0000 1111 : 1011 111w : mod reg r/m
memory8 to qwordregister (sign-extend)	0100 1RXB 0000 1111 : 1011 1110 : mod qwordreg r/m
memory16 to qwordregister	0100 1RXB 0000 1111 : 1011 1111 : mod qwordreg r/m
memory32 to qwordregister	0100 1RXB 0110 0011 : mod qwordreg r/m
MOVZX – Move with Zero-Extend	

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
register2 to register1	0100 0R0B : 0000 1111 : 1011 011w : 11 reg1 reg2
dwordregister2 to qwordregister1	0100 1R0B 0000 1111 : 1011 0111 : 11 qwordreg1 dwordreg2
memory to register	0100 0RXB : 0000 1111 : 1011 011w : mod reg r/m
memory32 to qwordregister	0100 1RXB 0000 1111 : 1011 0111 : mod qwordreg r/m
MUL – Unsigned Multiply	
AL, AX, or EAX with register	0100 000B : 1111 011w : 11 100 reg
RAX with qwordregister (to RDX:RAX)	0100 100B 1111 0111 : 11 100 qwordreg
AL, AX, or EAX with memory	0100 00XB 1111 011w : mod 100 r/m
RAX with memory64 (to RDX:RAX)	0100 10XB 1111 0111 : mod 100 r/m
NEG – Two's Complement Negation	
register	0100 000B : 1111 011w : 11 011 reg
qwordregister	0100 100B 1111 0111 : 11 011 qwordreg
memory	0100 00XB : 1111 011w : mod 011 r/m
memory64	0100 10XB 1111 0111 : mod 011 r/m
NOP – No Operation	1001 0000
NOT – One's Complement Negation	
register	0100 000B : 1111 011w : 11 010 reg
qwordregister	0100 000B 1111 0111 : 11 010 qwordreg
memory	0100 00XB : 1111 011w : mod 010 r/m
memory64	0100 1RXB 1111 0111 : mod 010 r/m
OR – Logical Inclusive OR	
register1 to register2	0000 100w : 11 reg1 reg2
byteregister1 to byteregister2	0100 0R0B 0000 1000 : 11 bytereg1 bytereg2
qwordregister1 to qwordregister2	0100 1R0B 0000 1001 : 11 qwordreg1 qwordreg2
register2 to register1	0000 101w : 11 reg1 reg2
byteregister2 to byteregister1	0100 0R0B 0000 1010 : 11 bytereg1 bytereg2
qwordregister2 to qwordregister1	0100 0R0B 0000 1011 : 11 qwordreg1 qwordreg2
memory to register	0000 101w : mod reg r/m
memory8 to byteregister	0100 0RXB 0000 1010 : mod bytereg r/m
memory8 to qwordregister	0100 0RXB 0000 1011 : mod qwordreg r/m
register to memory	0000 100w : mod reg r/m
byteregister to memory8	0100 0RXB 0000 1000 : mod bytereg r/m
qwordregister to memory64	0100 1RXB 0000 1001 : mod qwordreg r/m
immediate to register	1000 00sw : 11 001 reg : imm
immediate8 to byteregister	0100 000B 1000 0000 : 11 001 bytereg : imm8
immediate32 to qwordregister	0100 000B 1000 0001 : 11 001 qwordreg : imm32
immediate8 to qwordregister	0100 000B 1000 0011 : 11 001 qwordreg : imm8
immediate to AL, AX, or EAX	0000 110w : imm

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
immediate64 to RAX	0100 1000 0000 1101 : imm64
immediate to memory	1000 00sw : mod 001 r/m : imm
immediate8 to memory8	0100 00XB 1000 0000 : mod 001 r/m : imm8
immediate32 to memory64	0100 00XB 1000 0001 : mod 001 r/m : imm32
immediate8 to memory64	0100 00XB 1000 0011 : mod 001 r/m : imm8
OUT – Output to Port	
fixed port	1110 011w : port number
variable port	1110 111w
OUTS – Output to DX Port	
output to DX Port	0110 111w
POP – Pop a Value from the Stack	
wordregister	0101 0101 : 0100 000B : 1000 1111 : 11 000 reg16
qwordregister	0100 W00B[S] : 1000 1111 : 11 000 reg64
wordregister (alternate encoding)	0101 0101 : 0100 000B : 0101 1 reg16
qwordregister (alternate encoding)	0100 W00B : 0101 1 reg64
memory64	0100 W0XB[S] : 1000 1111 : mod 000 r/m
memory16	0101 0101 : 0100 00XB 1000 1111 : mod 000 r/m
POP – Pop a Segment Register from the Stack (Note: CS cannot be sreg2 in this usage.)	
segment register FS, GS	0000 1111: 10 sreg3 001
POPF/POPFQ – Pop Stack into FLAGS/RFLAGS Register	
pop stack to FLAGS register	0101 0101 : 1001 1101
pop Stack to RFLAGS register	0100 1000 1001 1101
PUSH – Push Operand onto the Stack	
wordregister	0101 0101 : 0100 000B : 1111 1111 : 11 110 reg16
qwordregister	0100 W00B[S] : 1111 1111 : 11 110 reg64
wordregister (alternate encoding)	0101 0101 : 0100 000B : 0101 0 reg16
qwordregister (alternate encoding)	0100 W00B[S] : 0101 0 reg64
memory16	0101 0101 : 0100 000B : 1111 1111 : mod 110 r/m
memory64	0100 W00B[S] : 1111 1111 : mod 110 r/m
immediate8	0110 1010 : imm8
immediate16	0101 0101 : 0110 1000 : imm16
immediate64	0110 1000 : imm64
PUSH – Push Segment Register onto the Stack	
segment register FS,GS	0000 1111: 10 sreg3 000
PUSHF/PUSHFD – Push Flags Register onto the Stack	1001 1100
RCL – Rotate thru Carry Left	
register by 1	0100 000B : 1101 000w : 11 010 reg
qwordregister by 1	0100 100B 1101 0001 : 11 010 qwordreg

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
memory by 1	0100 00XB : 1101 000w : mod 010 r/m
memory64 by 1	0100 10XB 1101 0001 : mod 010 r/m
register by CL	0100 000B : 1101 001w : 11 010 reg
qwordregister by CL	0100 100B 1101 0011 : 11 010 qwordreg
memory by CL	0100 00XB : 1101 001w : mod 010 r/m
memory64 by CL	0100 10XB 1101 0011 : mod 010 r/m
register by immediate count	0100 000B : 1100 000w : 11 010 reg : imm
qwordregister by immediate count	0100 100B 1100 0001 : 11 010 qwordreg : imm8
memory by immediate count	0100 00XB : 1100 000w : mod 010 r/m : imm
memory64 by immediate count	0100 10XB 1100 0001 : mod 010 r/m : imm8
RCR – Rotate thru Carry Right	
register by 1	0100 000B : 1101 000w : 11 011 reg
qwordregister by 1	0100 100B 1101 0001 : 11 011 qwordreg
memory by 1	0100 00XB : 1101 000w : mod 011 r/m
memory64 by 1	0100 10XB 1101 0001 : mod 011 r/m
register by CL	0100 000B : 1101 001w : 11 011 reg
qwordregister by CL	0100 000B 1101 0010 : 11 011 qwordreg
memory by CL	0100 00XB : 1101 001w : mod 011 r/m
memory64 by CL	0100 10XB 1101 0011 : mod 011 r/m
register by immediate count	0100 000B : 1100 000w : 11 011 reg : imm8
qwordregister by immediate count	0100 100B 1100 0001 : 11 011 qwordreg : imm8
memory by immediate count	0100 00XB : 1100 000w : mod 011 r/m : imm8
memory64 by immediate count	0100 10XB 1100 0001 : mod 011 r/m : imm8
RDMSR – Read from Model-Specific Register	
load ECX-specified register into EDX:EAX	0000 1111 : 0011 0010
RDPMC – Read Performance Monitoring Counters	
load ECX-specified performance counter into EDX:EAX	0000 1111 : 0011 0011
RDTSC – Read Time-Stamp Counter	
read time-stamp counter into EDX:EAX	0000 1111 : 0011 0001
RDTSCP – Read Time-Stamp Counter and Processor ID	0000 1111 : 0000 0001: 1111 1001
REP INS – Input String	
REP LODS – Load String	
REP MOVS – Move String	
REP OUTS – Output String	
REP STOS – Store String	
REPE CMPS – Compare String	
REPE SCAS – Scan String	
REPNE CMPS – Compare String	

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
REPNE SCAS – Scan String	
RET – Return from Procedure (to same segment)	
no argument	1100 0011
adding immediate to SP	1100 0010 : 16-bit displacement
RET – Return from Procedure (to other segment)	
intersegment	1100 1011
adding immediate to SP	1100 1010 : 16-bit displacement
ROL – Rotate Left	
register by 1	0100 000B 1101 000w : 11 000 reg
byteregister by 1	0100 000B 1101 0000 : 11 000 bytereg
qwordregister by 1	0100 100B 1101 0001 : 11 000 qwordreg
memory by 1	0100 00XB 1101 000w : mod 000 r/m
memory8 by 1	0100 00XB 1101 0000 : mod 000 r/m
memory64 by 1	0100 10XB 1101 0001 : mod 000 r/m
register by CL	0100 000B 1101 001w : 11 000 reg
byteregister by CL	0100 000B 1101 0010 : 11 000 bytereg
qwordregister by CL	0100 100B 1101 0011 : 11 000 qwordreg
memory by CL	0100 00XB 1101 001w : mod 000 r/m
memory8 by CL	0100 00XB 1101 0010 : mod 000 r/m
memory64 by CL	0100 10XB 1101 0011 : mod 000 r/m
register by immediate count	1100 000w : 11 000 reg : imm8
byteregister by immediate count	0100 000B 1100 0000 : 11 000 bytereg : imm8
qwordregister by immediate count	0100 100B 1100 0001 : 11 000 bytereg : imm8
memory by immediate count	1100 000w : mod 000 r/m : imm8
memory8 by immediate count	0100 00XB 1100 0000 : mod 000 r/m : imm8
memory64 by immediate count	0100 10XB 1100 0001 : mod 000 r/m : imm8
ROR – Rotate Right	
register by 1	0100 000B 1101 000w : 11 001 reg
byteregister by 1	0100 000B 1101 0000 : 11 001 bytereg
qwordregister by 1	0100 100B 1101 0001 : 11 001 qwordreg
memory by 1	0100 00XB 1101 000w : mod 001 r/m
memory8 by 1	0100 00XB 1101 0000 : mod 001 r/m
memory64 by 1	0100 10XB 1101 0001 : mod 001 r/m
register by CL	0100 000B 1101 001w : 11 001 reg
byteregister by CL	0100 000B 1101 0010 : 11 001 bytereg
qwordregister by CL	0100 100B 1101 0011 : 11 001 qwordreg
memory by CL	0100 00XB 1101 001w : mod 001 r/m
memory8 by CL	0100 00XB 1101 0010 : mod 001 r/m

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
memory64 by CL	0100 10XB 1101 0011 : mod 001 r/m
register by immediate count	0100 000B 1100 000w : 11 001 reg : imm8
byteregister by immediate count	0100 000B 1100 0000 : 11 001 reg : imm8
qwordregister by immediate count	0100 100B 1100 0001 : 11 001 qwordreg : imm8
memory by immediate count	0100 00XB 1100 000w : mod 001 r/m : imm8
memory8 by immediate count	0100 00XB 1100 0000 : mod 001 r/m : imm8
memory64 by immediate count	0100 10XB 1100 0001 : mod 001 r/m : imm8
RSM – Resume from System Management Mode	0000 1111 : 1010 1010
SAL – Shift Arithmetic Left	same instruction as SHL
SAR – Shift Arithmetic Right	
register by 1	0100 000B 1101 000w : 11 111 reg
byteregister by 1	0100 000B 1101 0000 : 11 111 bytereg
qwordregister by 1	0100 100B 1101 0001 : 11 111 qwordreg
memory by 1	0100 00XB 1101 000w : mod 111 r/m
memory8 by 1	0100 00XB 1101 0000 : mod 111 r/m
memory64 by 1	0100 10XB 1101 0001 : mod 111 r/m
register by CL	0100 000B 1101 001w : 11 111 reg
byteregister by CL	0100 000B 1101 0010 : 11 111 bytereg
qwordregister by CL	0100 100B 1101 0011 : 11 111 qwordreg
memory by CL	0100 00XB 1101 001w : mod 111 r/m
memory8 by CL	0100 00XB 1101 0010 : mod 111 r/m
memory64 by CL	0100 10XB 1101 0011 : mod 111 r/m
register by immediate count	0100 000B 1100 000w : 11 111 reg : imm8
byteregister by immediate count	0100 000B 1100 0000 : 11 111 bytereg : imm8
qwordregister by immediate count	0100 100B 1100 0001 : 11 111 qwordreg : imm8
memory by immediate count	0100 00XB 1100 000w : mod 111 r/m : imm8
memory8 by immediate count	0100 00XB 1100 0000 : mod 111 r/m : imm8
memory64 by immediate count	0100 10XB 1100 0001 : mod 111 r/m : imm8
SBB – Integer Subtraction with Borrow	
register1 to register2	0100 0R0B 0001 100w : 11 reg1 reg2
byteregister1 to byteregister2	0100 0R0B 0001 1000 : 11 bytereg1 bytereg2
quadregister1 to quadregister2	0100 1R0B 0001 1001 : 11 quadreg1 quadreg2
register2 to register1	0100 0R0B 0001 101w : 11 reg1 reg2
byteregister2 to byteregister1	0100 0R0B 0001 1010 : 11 reg1 bytereg2
byteregister2 to byteregister1	0100 1R0B 0001 1011 : 11 reg1 bytereg2
memory to register	0100 0RXB 0001 101w : mod reg r/m
memory8 to byteregister	0100 0RXB 0001 1010 : mod bytereg r/m
memory64 to byteregister	0100 1RXB 0001 1011 : mod quadreg r/m

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
register to memory	0100 0RXB 0001 100w : mod reg r/m
byteregister to memory8	0100 0RXB 0001 1000 : mod reg r/m
quadregister to memory64	0100 1RXB 0001 1001 : mod reg r/m
immediate to register	0100 000B 1000 00sw : 11 011 reg : imm
immediate8 to byteregister	0100 000B 1000 0000 : 11 011 bytereg : imm8
immediate32 to qwordregister	0100 100B 1000 0001 : 11 011 qwordreg : imm32
immediate8 to qwordregister	0100 100B 1000 0011 : 11 011 qwordreg : imm8
immediate to AL, AX, or EAX	0100 000B 0001 110w : imm
immediate32 to RAL	0100 1000 0001 1101 : imm32
immediate to memory	0100 00XB 1000 00sw : mod 011 r/m : imm
immediate8 to memory8	0100 00XB 1000 0000 : mod 011 r/m : imm8
immediate32 to memory64	0100 10XB 1000 0001 : mod 011 r/m : imm32
immediate8 to memory64	0100 10XB 1000 0011 : mod 011 r/m : imm8
SCAS/SCASB/SCASW/SCASD – Scan String	
scan string	1010 111w
scan string (compare AL with byte at RDI)	0100 1000 1010 1110
scan string (compare RAX with qword at RDI)	0100 1000 1010 1111
SETcc – Byte Set on Condition	
register	0100 000B 0000 1111 : 1001 tttn : 11 000 reg
register	0100 0000 0000 1111 : 1001 tttn : 11 000 reg
memory	0100 00XB 0000 1111 : 1001 tttn : mod 000 r/m
memory	0100 0000 0000 1111 : 1001 tttn : mod 000 r/m
SGDT – Store Global Descriptor Table Register	0000 1111 : 0000 0001 : mod[A] 000 r/m
SHL – Shift Left	
register by 1	0100 000B 1101 000w : 11 100 reg
byteregister by 1	0100 000B 1101 0000 : 11 100 bytereg
qwordregister by 1	0100 100B 1101 0001 : 11 100 qwordreg
memory by 1	0100 00XB 1101 000w : mod 100 r/m
memory8 by 1	0100 00XB 1101 0000 : mod 100 r/m
memory64 by 1	0100 10XB 1101 0001 : mod 100 r/m
register by CL	0100 000B 1101 001w : 11 100 reg
byteregister by CL	0100 000B 1101 0010 : 11 100 bytereg
qwordregister by CL	0100 100B 1101 0011 : 11 100 qwordreg
memory by CL	0100 00XB 1101 001w : mod 100 r/m
memory8 by CL	0100 00XB 1101 0010 : mod 100 r/m
memory64 by CL	0100 10XB 1101 0011 : mod 100 r/m
register by immediate count	0100 000B 1100 000w : 11 100 reg : imm8
byteregister by immediate count	0100 000B 1100 0000 : 11 100 bytereg : imm8

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
quadregister by immediate count	0100 100B 1100 0001 : 11 100 quadreg : imm8
memory by immediate count	0100 00XB 1100 000w : mod 100 r/m : imm8
memory8 by immediate count	0100 00XB 1100 0000 : mod 100 r/m : imm8
memory64 by immediate count	0100 10XB 1100 0001 : mod 100 r/m : imm8
SHLD – Double Precision Shift Left	
register by immediate count	0100 0R0B 0000 1111 : 1010 0100 : 11 reg2 reg1 : imm8
qwordregister by immediate8	0100 1R0B 0000 1111 : 1010 0100 : 11 qworddreg2 qwordreg1 : imm8
memory by immediate count	0100 0RXB 0000 1111 : 1010 0100 : mod reg r/m : imm8
memory64 by immediate8	0100 1RXB 0000 1111 : 1010 0100 : mod qwordreg r/m : imm8
register by CL	0100 0R0B 0000 1111 : 1010 0101 : 11 reg2 reg1
quadregister by CL	0100 1R0B 0000 1111 : 1010 0101 : 11 quadreg2 quadreg1
memory by CL	0100 00XB 0000 1111 : 1010 0101 : mod reg r/m
memory64 by CL	0100 1RXB 0000 1111 : 1010 0101 : mod quadreg r/m
SHR – Shift Right	
register by 1	0100 000B 1101 000w : 11 101 reg
byteregister by 1	0100 000B 1101 0000 : 11 101 bytereg
qwordregister by 1	0100 100B 1101 0001 : 11 101 qwordreg
memory by 1	0100 00XB 1101 000w : mod 101 r/m
memory8 by 1	0100 00XB 1101 0000 : mod 101 r/m
memory64 by 1	0100 10XB 1101 0001 : mod 101 r/m
register by CL	0100 000B 1101 001w : 11 101 reg
byteregister by CL	0100 000B 1101 0010 : 11 101 bytereg
qwordregister by CL	0100 100B 1101 0011 : 11 101 qwordreg
memory by CL	0100 00XB 1101 001w : mod 101 r/m
memory8 by CL	0100 00XB 1101 0010 : mod 101 r/m
memory64 by CL	0100 10XB 1101 0011 : mod 101 r/m
register by immediate count	0100 000B 1100 000w : 11 101 reg : imm8
byteregister by immediate count	0100 000B 1100 0000 : 11 101 reg : imm8
qwordregister by immediate count	0100 100B 1100 0001 : 11 101 reg : imm8
memory by immediate count	0100 00XB 1100 000w : mod 101 r/m : imm8
memory8 by immediate count	0100 00XB 1100 0000 : mod 101 r/m : imm8
memory64 by immediate count	0100 10XB 1100 0001 : mod 101 r/m : imm8
SHRD – Double Precision Shift Right	
register by immediate count	0100 0R0B 0000 1111 : 1010 1100 : 11 reg2 reg1 : imm8
qwordregister by immediate8	0100 1R0B 0000 1111 : 1010 1100 : 11 qwordreg2 qwordreg1 : imm8
memory by immediate count	0100 00XB 0000 1111 : 1010 1100 : mod reg r/m : imm8

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
memory64 by immediate8	0100 1RXB 0000 1111 : 1010 1100 : mod qwordreg r/m : imm8
register by CL	0100 000B 0000 1111 : 1010 1101 : 11 reg2 reg1
qwordregister by CL	0100 1R0B 0000 1111 : 1010 1101 : 11 qwordreg2 qwordreg1
memory by CL	0000 1111 : 1010 1101 : mod reg r/m
memory64 by CL	0100 1RXB 0000 1111 : 1010 1101 : mod qwordreg r/m
SIDT – Store Interrupt Descriptor Table Register	0000 1111 : 0000 0001 : mod[A] 001 r/m
SLDT – Store Local Descriptor Table Register	
to register	0100 000B 0000 1111 : 0000 0000 : 11 000 reg
to memory	0100 00XB 0000 1111 : 0000 0000 : mod 000 r/m
SMSW – Store Machine Status Word	
to register	0100 000B 0000 1111 : 0000 0001 : 11 100 reg
to memory	0100 00XB 0000 1111 : 0000 0001 : mod 100 r/m
STC – Set Carry Flag	1111 1001
STD – Set Direction Flag	1111 1101
STI – Set Interrupt Flag	1111 1011
STOS/STOSB/STOSW/STOSD/STOSQ – Store String Data	
store string data	1010 101w
store string data (RAX at address RDI)	0100 1000 1010 1011
STR – Store Task Register	
to register	0100 000B 0000 1111 : 0000 0000 : 11 001 reg
to memory	0100 00XB 0000 1111 : 0000 0000 : mod 001 r/m
SUB – Integer Subtraction	
register1 from register2	0100 0R0B 0010 100w : 11 reg1 reg2
byteregister1 from byteregister2	0100 0R0B 0010 1000 : 11 bytereg1 bytereg2
qwordregister1 from qwordregister2	0100 1R0B 0010 1000 : 11 qwordreg1 qwordreg2
register2 from register1	0100 0R0B 0010 101w : 11 reg1 reg2
byteregister2 from byteregister1	0100 0R0B 0010 1010 : 11 bytereg1 bytereg2
qwordregister2 from qwordregister1	0100 1R0B 0010 1011 : 11 qwordreg1 qwordreg2
memory from register	0100 00XB 0010 101w : mod reg r/m
memory8 from byteregister	0100 0RXB 0010 1010 : mod bytereg r/m
memory64 from qwordregister	0100 1RXB 0010 1011 : mod qwordreg r/m
register from memory	0100 0RXB 0010 100w : mod reg r/m
byteregister from memory8	0100 0RXB 0010 1000 : mod bytereg r/m
qwordregister from memory8	0100 1RXB 0010 1000 : mod qwordreg r/m
immediate from register	0100 000B 1000 00sw : 11 101 reg : imm
immediate8 from byteregister	0100 000B 1000 0000 : 11 101 bytereg : imm8
immediate32 from qwordregister	0100 100B 1000 0001 : 11 101 qwordreg : imm32

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
immediate8 from qwordregister	0100 100B 1000 0011 : 11 101 qwordreg : imm8
immediate from AL, AX, or EAX	0100 000B 0010 110w : imm
immediate32 from RAX	0100 1000 0010 1101 : imm32
immediate from memory	0100 00XB 1000 00sw : mod 101 r/m : imm
immediate8 from memory8	0100 00XB 1000 0000 : mod 101 r/m : imm8
immediate32 from memory64	0100 10XB 1000 0001 : mod 101 r/m : imm32
immediate8 from memory64	0100 10XB 1000 0011 : mod 101 r/m : imm8
SWAPGS – Swap GS Base Register	
Exchanges the current GS base register value for value in MSR C0000102H	0000 1111 0000 0001 1111 1000
SYSCALL – Fast System Call	
fast call to privilege level 0 system procedures	0000 1111 0000 0101
SYSRET – Return From Fast System Call	
return from fast system call	0000 1111 0000 0111
TEST – Logical Compare	
register1 and register2	0100 0R0B 1000 010w : 11 reg1 reg2
byteregister1 and byteregister2	0100 0R0B 1000 0100 : 11 bytereg1 bytereg2
qwordregister1 and qwordregister2	0100 1R0B 1000 0101 : 11 qwordreg1 qwordreg2
memory and register	0100 0R0B 1000 010w : mod reg r/m
memory8 and byteregister	0100 0RXB 1000 0100 : mod bytereg r/m
memory64 and qwordregister	0100 1RXB 1000 0101 : mod qwordreg r/m
immediate and register	0100 000B 1111 011w : 11 000 reg : imm
immediate8 and byteregister	0100 000B 1111 0110 : 11 000 bytereg : imm8
immediate32 and qwordregister	0100 100B 1111 0111 : 11 000 bytereg : imm8
immediate and AL, AX, or EAX	0100 000B 1010 100w : imm
immediate32 and RAX	0100 1000 1010 1001 : imm32
immediate and memory	0100 00XB 1111 011w : mod 000 r/m : imm
immediate8 and memory8	0100 1000 1111 0110 : mod 000 r/m : imm8
immediate32 and memory64	0100 1000 1111 0111 : mod 000 r/m : imm32
UD2 – Undefined instruction	0000 FFFF : 0000 1011
VERR – Verify a Segment for Reading	
register	0100 000B 0000 1111 : 0000 0000 : 11 100 reg
memory	0100 00XB 0000 1111 : 0000 0000 : mod 100 r/m
VERW – Verify a Segment for Writing	
register	0100 000B 0000 1111 : 0000 0000 : 11 101 reg
memory	0100 00XB 0000 1111 : 0000 0000 : mod 101 r/m
WAIT – Wait	1001 1011
WBINVD – Writeback and Invalidate Data Cache	0000 1111 : 0000 1001

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
WRMSR – Write to Model-Specific Register	
write EDX:EAX to ECX specified MSR	0000 1111 : 0011 0000
write RDX[31:0]:RAX[31:0] to RCX specified MSR	0100 1000 0000 1111 : 0011 0000
XADD – Exchange and Add	
register1, register2	0100 0R0B 0000 1111 : 1100 000w : 11 reg2 reg1
byteregister1, byteregister2	0100 0R0B 0000 1111 : 1100 0000 : 11 bytereg2 bytereg1
qwordregister1, qwordregister2	0100 0R0B 0000 1111 : 1100 0001 : 11 qwordreg2 qwordreg1
memory, register	0100 0RXB 0000 1111 : 1100 000w : mod reg r/m
memory8, bytereg	0100 1RXB 0000 1111 : 1100 0000 : mod bytereg r/m
memory64, qwordreg	0100 1RXB 0000 1111 : 1100 0001 : mod qwordreg r/m
XCHG – Exchange Register/Memory with Register	
register1 with register2	1000 011w : 11 reg1 reg2
AX or EAX with register	1001 0 reg
memory with register	1000 011w : mod reg r/m
XLAT/XLATB – Table Look-up Translation	
AL to byte DS:[(E)BX + unsigned AL]	1101 0111
AL to byte DS:[RBX + unsigned AL]	0100 1000 1101 0111
XOR – Logical Exclusive OR	
register1 to register2	0100 0RXB 0011 000w : 11 reg1 reg2
byteregister1 to byteregister2	0100 0R0B 0011 0000 : 11 bytereg1 bytereg2
qwordregister1 to qwordregister2	0100 1R0B 0011 0001 : 11 qwordreg1 qwordreg2
register2 to register1	0100 0R0B 0011 001w : 11 reg1 reg2
byteregister2 to byteregister1	0100 0R0B 0011 0010 : 11 bytereg1 bytereg2
qwordregister2 to qwordregister1	0100 1R0B 0011 0011 : 11 qwordreg1 qwordreg2
memory to register	0100 0RXB 0011 001w : mod reg r/m
memory8 to byteregister	0100 0RXB 0011 0010 : mod bytereg r/m
memory64 to qwordregister	0100 1RXB 0011 0011 : mod qwordreg r/m
register to memory	0100 0RXB 0011 000w : mod reg r/m
byteregister to memory8	0100 0RXB 0011 0000 : mod bytereg r/m
qwordregister to memory8	0100 1RXB 0011 0001 : mod qwordreg r/m
immediate to register	0100 000B 1000 00sw : 11 110 reg : imm
immediate8 to byteregister	0100 000B 1000 0000 : 11 110 bytereg : imm8
immediate32 to qwordregister	0100 100B 1000 0001 : 11 110 qwordreg : imm32
immediate8 to qwordregister	0100 100B 1000 0011 : 11 110 qwordreg : imm8
immediate to AL, AX, or EAX	0100 000B 0011 010w : imm
immediate to RAX	0100 1000 0011 0101 : immediate data
immediate to memory	0100 00XB 1000 00sw : mod 110 r/m : imm
immediate8 to memory8	0100 00XB 1000 0000 : mod 110 r/m : imm8

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
immediate32 to memory64	0100 10XB 1000 0001 : mod 110 r/m : imm32
immediate8 to memory64	0100 10XB 1000 0011 : mod 110 r/m : imm8
Prefix Bytes	
address size	0110 0111
LOCK	1111 0000
operand size	0110 0110
CS segment override	0010 1110
DS segment override	0011 1110
ES segment override	0010 0110
FS segment override	0110 0100
GS segment override	0110 0101
SS segment override	0011 0110

B.3 PENTIUM® PROCESSOR FAMILY INSTRUCTION FORMATS AND ENCODINGS

The following table shows formats and encodings introduced by the Pentium processor family.

Table B-16. Pentium® Processor Family Instruction Formats and Encodings, Non-64-Bit Modes

Instruction and Format	Encoding
CMPXCHG8B – Compare and Exchange 8 Bytes	
EDX:EAX with memory64	0000 1111 : 1100 0111 : mod 001 r/m

Table B-17. Pentium® Processor Family Instruction Formats and Encodings, 64-Bit Mode

Instruction and Format	Encoding
CMPXCHG8B/CMPXCHG16B – Compare and Exchange Bytes	
EDX:EAX with memory64	0000 1111 : 1100 0111 : mod 001 r/m
RDX:RAX with memory128	0100 10XB 0000 1111 : 1100 0111 : mod 001 r/m

B.4 64-BIT MODE INSTRUCTION ENCODINGS FOR SIMD INSTRUCTION EXTENSIONS

Non-64-bit mode instruction encodings for MMX Technology, SSE, SSE2, and SSE3 are covered by applying these rules to Table B-19 through Table B-31. Table B-34 lists special encodings (instructions that do not follow the rules below).

1. The REX instruction has no effect:
    - On immediates.
    - If both operands are MMX registers.
    - On MMX registers and XMM registers.
    - If an MMX register is encoded in the reg field of the ModR/M byte.

2. If a memory operand is encoded in the r/m field of the ModR/M byte, REX.X and REX.B may be used for encoding the memory operand.
3. If a general-purpose register is encoded in the r/m field of the ModR/M byte, REX.B may be used for register encoding and REX.W may be used to encode the 64-bit operand size.
4. If an XMM register operand is encoded in the reg field of the ModR/M byte, REX.R may be used for register encoding. If an XMM register operand is encoded in the r/m field of the ModR/M byte, REX.B may be used for register encoding.

B.5 MMX INSTRUCTION FORMATS AND ENCODINGS

MMX instructions, except the EMMS instruction, use a format similar to the 2-byte Intel Architecture integer format. Details of subfield encodings within these formats are presented below.

B.5.1 Granularity Field (gg)

The granularity field (gg) indicates the size of the packed operands that the instruction is operating on. When this field is used, it is located in bits 1 and 0 of the second opcode byte. Table B-18 shows the encoding of the gg field.

Table B-18. Encoding of Granularity of Data Field (gg)

gg	Granularity of Data
00	Packed Bytes
01	Packed Words
10	Packed Doublewords
11	Quadword

B.5.2 MMX Technology and General-Purpose Register Fields (mmxreg and reg)

When MMX technology registers (mmxreg) are used as operands, they are encoded in the ModR/M byte in the reg field (bits 5, 4, and 3) and/or the R/M field (bits 2, 1, and 0).

If an MMX instruction operates on a general-purpose register (reg), the register is encoded in the R/M field of the ModR/M byte.

B.5.3 MMX Instruction Formats and Encodings Table

Table B-19 shows the formats and encodings of the integer instructions.

Table B-19. MMX Instruction Formats and Encodings

Instruction and Format	Encoding
EMMS – Empty MMX technology state	0000 1111:01110111
MOVD – Move doubleword	
reg to mmxreg	0000 1111:0110 1110: 11 mmxreg reg
reg from mmxreg	0000 1111:0111 1110: 11 mmxreg reg
mem to mmxreg	0000 1111:0110 1110: mod mmxreg r/m
mem from mmxreg	0000 1111:0111 1110: mod mmxreg r/m
MOVQ – Move quadword	
mmxreg2 to mmxreg1	0000 1111:0110 1111: 11 mmxreg1 mmxreg2
mmxreg2 from mmxreg1	0000 1111:0111 1111: 11 mmxreg1 mmxreg2

Table B-19. MMX Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
mem to mmxreg	0000 1111:0110 1111: mod mmxreg r/m
mem from mmxreg	0000 1111:0111 1111: mod mmxreg r/m
PACKSSDW[1] – Pack dword to word data (signed with saturation)	
mmxreg2 to mmxreg1	0000 1111:0110 1011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 1011: mod mmxreg r/m
PACKSSWB[1] – Pack word to byte data (signed with saturation)	
mmxreg2 to mmxreg1	0000 1111:0110 0011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 0011: mod mmxreg r/m
PACKUSWB[1] – Pack word to byte data (unsigned with saturation)	
mmxreg2 to mmxreg1	0000 1111:0110 0111: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 0111: mod mmxreg r/m
PADD – Add with wrap-around	
mmxreg2 to mmxreg1	0000 1111: 1111 11gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 1111 11gg: mod mmxreg r/m
PADDS – Add signed with saturation	
mmxreg2 to mmxreg1	0000 1111: 1110 11gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 1110 11gg: mod mmxreg r/m
PADDUS – Add unsigned with saturation	
mmxreg2 to mmxreg1	0000 1111: 1101 11gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 1101 11gg: mod mmxreg r/m
PAND – Bitwise And	
mmxreg2 to mmxreg1	0000 1111:1101 1011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1101 1011: mod mmxreg r/m
PANDN – Bitwise AndNot	
mmxreg2 to mmxreg1	0000 1111:1101 1111: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1101 1111: mod mmxreg r/m
PCMPEQ – Packed compare for equality	
mmxreg1 with mmxreg2	0000 1111:0111 01gg: 11 mmxreg1 mmxreg2
mmxreg with memory	0000 1111:0111 01gg: mod mmxreg r/m
PCMPGT – Packed compare greater (signed)	
mmxreg1 with mmxreg2	0000 1111:0110 01gg: 11 mmxreg1 mmxreg2
mmxreg with memory	0000 1111:0110 01gg: mod mmxreg r/m
PMADDWD – Packed multiply add	
mmxreg2 to mmxreg1	0000 1111:1111 0101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1111 0101: mod mmxreg r/m
PMULHUW – Packed multiplication, store high word (unsigned)	
mmxreg2 to mmxreg1	0000 1111: 1110 0100: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 1110 0100: mod mmxreg r/m

Table B-19. MMX Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
PMULHW – Packed multiplication, store high word	
mmxreg2 to mmxreg1	0000 1111:1110 0101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1110 0101: mod mmxreg r/m
PMULLW – Packed multiplication, store low word	
mmxreg2 to mmxreg1	0000 1111:1101 0101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1101 0101: mod mmxreg r/m
POR – Bitwise Or	
mmxreg2 to mmxreg1	0000 1111:1110 1011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1110 1011: mod mmxreg r/m
PSLL[2] – Packed shift left logical	
mmxreg1 by mmxreg2	0000 1111:1111 00gg: 11 mmxreg1 mmxreg2
mmxreg by memory	0000 1111:1111 00gg: mod mmxreg r/m
mmxreg by immediate	0000 1111:0111 00gg: 11 110 mmxreg: imm8 data
PSRA[2] – Packed shift right arithmetic	
mmxreg1 by mmxreg2	0000 1111:1110 00gg: 11 mmxreg1 mmxreg2
mmxreg by memory	0000 1111:1110 00gg: mod mmxreg r/m
mmxreg by immediate	0000 1111:0111 00gg: 11 100 mmxreg: imm8 data
PSRL[2] – Packed shift right logical	
mmxreg1 by mmxreg2	0000 1111:1101 00gg: 11 mmxreg1 mmxreg2
mmxreg by memory	0000 1111:1101 00gg: mod mmxreg r/m
mmxreg by immediate	0000 1111:0111 00gg: 11 010 mmxreg: imm8 data
PSUB – Subtract with wrap-around	
mmxreg2 from mmxreg1	0000 1111:1111 10gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:1111 10gg: mod mmxreg r/m
PSUBS – Subtract signed with saturation	
mmxreg2 from mmxreg1	0000 1111:1110 10gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:1110 10gg: mod mmxreg r/m
PSUBUS – Subtract unsigned with saturation	
mmxreg2 from mmxreg1	0000 1111:1101 10gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:1101 10gg: mod mmxreg r/m
PUNPCKH – Unpack high data to next larger type	
mmxreg2 to mmxreg1	0000 1111:0110 10gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 10gg: mod mmxreg r/m
PUNPCKL – Unpack low data to next larger type	
mmxreg2 to mmxreg1	0000 1111:0110 00gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 00gg: mod mmxreg r/m

Table B-19. MMX Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
PXOR – Bitwise Xor	
mmxreg2 to mmxreg1	0000 1111:1110 1111: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1110 1111: mod mmxreg r/m

NOTES:

1. The pack instructions perform saturation from signed packed data of one type to signed or unsigned data of the next smaller type.
2. The format of the shift instructions has one additional format to support shifting by immediate shift-counts. The shift operations are not supported equally for all data types.

B.6 PROCESSOR EXTENDED STATE INSTRUCTION FORMATS AND ENCODINGS

Table B-20 shows the formats and encodings for several instructions that relate to processor extended state management.

Table B-20. Formats and Encodings of XSAVE/XRSTOR/XGETBV/XSETBV Instructions

Instruction and Format	Encoding
XGETBV – Get Value of Extended Control Register	0000 1111:0000 0001: 1101 0000
XRSTOR – Restore Processor Extended States[1]	0000 1111:1010 1110: mod[A] 101 r/m
XSAVE – Save Processor Extended States[1]	0000 1111:1010 1110: mod[A] 100 r/m
XSETBV – Set Extended Control Register	0000 1111:0000 0001: 1101 0001

NOTES:

1. For XSAVE and XRSTOR, "mod = 11" is reserved.

B.7 P6 FAMILY INSTRUCTION FORMATS AND ENCODINGS

Table B-20 shows the formats and encodings for several instructions that were introduced into the IA-32 architecture in the P6 family processors.

Table B-21. Formats and Encodings of P6 Family Instructions

Instruction and Format	Encoding
CMOVcc – Conditional Move	
register2 to register1	0000 1111: 0100 tttn : 11 reg1 reg2
memory to register	0000 1111 : 0100 tttn : mod reg r/m
FCMOVcc – Conditional Move on EFLAG Register Condition Codes	
move if below (B)	11011 010 : 11 000 ST(i)
move if equal (E)	11011 010 : 11 001 ST(i)
move if below or equal (BE)	11011 010 : 11 010 ST(i)
move if unordered (U)	11011 010 : 11 011 ST(i)
move if not below (NB)	11011 011 : 11 000 ST(i)
move if not equal (NE)	11011 011 : 11 001 ST(i)
move if not below or equal (NBE)	11011 011 : 11 010 ST(i)

Table B-21. Formats and Encodings of P6 Family Instructions (Contd.)

Instruction and Format	Encoding
move if not unordered (NU)	11011 011 : 11 011 ST(i)
FCOMI – Compare Real and Set EFLAGS	11011 011 : 11 110 ST(i)
FXRSTOR – Restore x87 FPU, MMX, SSE, and SSE2 State[1]	0000 1111:1010 1110: mod[A] 001 r/m
FXSAVE – Save x87 FPU, MMX, SSE, and SSE2 State[1]	0000 1111:1010 1110: mod[A] 000 r/m
SYSENTER – Fast System Call	0000 1111:0011 0100
SYSEXIT – Fast Return from Fast System Call	0000 1111:0011 0101

NOTES:

1. For FXSAVE and FXRSTOR, "mod = 11" is reserved.

B.8 SSE INSTRUCTION FORMATS AND ENCODINGS

The SSE instructions use the ModR/M format and are preceded by the 0FH prefix byte. In general, operations are not duplicated to provide two directions (that is, separate load and store variants).

The following three tables (Tables B-22, B-23, and B-24) show the formats and encodings for the SSE SIMD floating-point, SIMD integer, and cacheability and memory ordering instructions, respectively. Some SSE instructions require a mandatory prefix (66H, F2H, F3H) as part of the two-byte opcode. Mandatory prefixes are included in the tables.

Table B-22. Formats and Encodings of SSE Floating-Point Instructions

Instruction and Format	Encoding
ADDPS—Add Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 1000:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1000: mod xmmreg r/m
ADDSS—Add Scalar Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	1111 0011:0000 1111:01011000:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:01011000: mod xmmreg r/m
ANDNPS—Bitwise Logical AND NOT of Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0101: mod xmmreg r/m
ANDPS—Bitwise Logical AND of Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0100: mod xmmreg r/m
CMPPS—Compare Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1, imm8	0000 1111:1100 0010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0000 1111:1100 0010: mod xmmreg r/m: imm8
CMPSS—Compare Scalar Single Precision Floating-Point Values	
xmmreg2 to xmmreg1, imm8	1111 0011:0000 1111:1100 0010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	1111 0011:0000 1111:1100 0010: mod xmmreg r/m: imm8
COMISS—Compare Scalar Ordered Single Precision Floating-Point Values and Set EFLAGS	
xmmreg2 to xmmreg1	0000 1111:0010 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0010 1111: mod xmmreg r/m

Table B-22. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
CVTPI2PS—Convert Packed Doubleword Integers to Packed Single Precision Floating-Point Values	
mmreg to xmmreg	0000 1111:0010 1010:11 xmmreg1 mmreg1
mem to xmmreg	0000 1111:0010 1010: mod xmmreg r/m
CVTPS2PI—Convert Packed Single Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to mmreg	0000 1111:0010 1101:11 mmreg1 xmmreg1
mem to mmreg	0000 1111:0010 1101: mod mmreg r/m
CVTSI2SS—Convert Signed Integer to Scalar Single Precision Floating-Point Value	
r32 to xmmreg1	1111 0011:0000 1111:00101010:11 xmmreg1 r32
mem to xmmreg	1111 0011:0000 1111:00101010: mod xmmreg r/m
CVTSS2SI—Convert Scalar Single Precision Floating-Point Value to Signed Integer	
xmmreg to r32	1111 0011:0000 1111:0010 1101:11 r32 xmmreg
mem to r32	1111 0011:0000 1111:0010 1101: mod r32 r/m
CVTTPS2PI—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to mmreg	0000 1111:0010 1100:11 mmreg1 xmmreg1
mem to mmreg	0000 1111:0010 1100: mod mmreg r/m
CVTTSS2SI—Convert with Truncation Scalar Single Precision Floating-Point Value to Signed Integer	
xmmreg to r32	1111 0011:0000 1111:0010 1100:11 r32 xmmreg1
mem to r32	1111 0011:0000 1111:0010 1100: mod r32 r/m
DIVPS—Divide Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1110: mod xmmreg r/m
DIVSS—Divide Scalar Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 1110:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 1110: mod xmmreg r/m
LDMXCSR—Load MXCSR Register State	
m32 to MXCSR	0000 1111:1010 1110:mod[A] 010 mem
MAXPS—Return Maximum Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1111: mod xmmreg r/m
MAXSS—Return Maximum Scalar Double Precision Floating-Point Value	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 1111:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 1111: mod xmmreg r/m
MINPS—Return Minimum Packed Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1101: mod xmmreg r/m
MINSS—Return Minimum Scalar Double Precision Floating-Point Value	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 1101:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 1101: mod xmmreg r/m

Table B-22. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
MOVAPS—Move Aligned Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0010 1000:11 xmmreg2 xmmreg1
mem to xmmreg1	0000 1111:0010 1000: mod xmmreg r/m
xmmreg1 to xmmreg2	0000 1111:0010 1001:11 xmmreg1 xmmreg2
xmmreg1 to mem	0000 1111:0010 1001: mod xmmreg r/m
MOVHLPS—Move Packed Single Precision Floating-Point Values High to Low	
xmmreg2 to xmmreg1	0000 1111:0001 0010:11 xmmreg1 xmmreg2
MOVHPS—Move High Packed Single Precision Floating-Point Values	
mem to xmmreg	0000 1111:0001 0110: mod xmmreg r/m
xmmreg to mem	0000 1111:0001 0111: mod xmmreg r/m
MOVLHPS—Move Packed Single Precision Floating-Point Values Low to High	
xmmreg2 to xmmreg1	0000 1111:00010110:11 xmmreg1 xmmreg2
MOVLPS—Move Low Packed Single Precision Floating-Point Values	
mem to xmmreg	0000 1111:0001 0010: mod xmmreg r/m
xmmreg to mem	0000 1111:0001 0011: mod xmmreg r/m
MOVMSKPS—Extract Packed Single Precision Floating-Point Sign Mask	
xmmreg to r32	0000 1111:0101 0000:11 r32 xmmreg
MOVSS—Move Scalar Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0001 0000:11 xmmreg2 xmmreg1
mem to xmmreg1	1111 0011:0000 1111:0001 0000: mod xmmreg r/m
xmmreg1 to xmmreg2	1111 0011:0000 1111:0001 0001:11 xmmreg1 xmmreg2
xmmreg1 to mem	1111 0011:0000 1111:0001 0001: mod xmmreg r/m
MOVUPS—Move Unaligned Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0001 0000:11 xmmreg2 xmmreg1
mem to xmmreg1	0000 1111:0001 0000: mod xmmreg r/m
xmmreg1 to xmmreg2	0000 1111:0001 0001:11 xmmreg1 xmmreg2
xmmreg1 to mem	0000 1111:0001 0001: mod xmmreg r/m
MULPS—Multiply Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 1001:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1001: mod xmmreg r/m
MULSS—Multiply Scalar Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 1001:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 1001: mod xmmreg r/m
ORPS—Bitwise Logical OR of Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 0110:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0110: mod xmmreg r/m
RCPPS—Compute Reciprocals of Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 0011:11 xmmreg1 xmmreg2

Table B-22. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
mem to xmmreg	0000 1111:0101 0011: mod xmmreg r/m
RCPSS—Compute Reciprocals of Scalar Single Precision Floating-Point Value	
xmmreg2 to xmmreg1	1111 0011:0000 1111:01010011:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:01010011: mod xmmreg r/m
RSQRTPS—Compute Reciprocals of Square Roots of Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 0010:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0010: mode xmmreg r/m
RSQRTSS—Compute Reciprocals of Square Roots of Scalar Single Precision Floating-Point Value	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 0010:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 0010: mod xmmreg r/m
SHUFPS—Shuffle Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1, imm8	0000 1111:1100 0110:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0000 1111:1100 0110: mod xmmreg r/m: imm8
SQRTPS—Compute Square Roots of Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 0001:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0001: mod xmmreg r/m
SQRTSS—Compute Square Root of Scalar Single Precision Floating-Point Value	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 0001:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 0001:mod xmmreg r/m
STMXCSR—Store MXCSR Register State	
MXCSR to mem	0000 1111:1010 1110:mod[A] 011 mem
SUBPS—Subtract Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 1100:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1100:mod xmmreg r/m
SUBSS—Subtract Scalar Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 1100:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 1100:mod xmmreg r/m
UCOMISS—Unordered Compare Scalar Ordered Single Precision Floating-Point Values and Set EFLAGS	
xmmreg2 to xmmreg1	0000 1111:0010 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0010 1110: mod xmmreg r/m
UNPCKHPS—Unpack and Interleave High Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0001 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0001 0101: mod xmmreg r/m
UNPCKLPS—Unpack and Interleave Low Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0001 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0001 0100: mod xmmreg r/m
XORPS—Bitwise Logical XOR of Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 0111:11 xmmreg1 xmmreg2

Table B-22. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
mem to xmmreg	0000 1111:0101 0111: mod xmmreg r/m

Table B-23. Formats and Encodings of SSE Integer Instructions

Instruction and Format	Encoding
PAVGB/PAVGW—Average Packed Integers	
mmreg2 to mmreg1	0000 1111:1110 0000:11 mmreg1 mmreg2
	0000 1111:1110 0011:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1110 0000: mod mmreg r/m
	0000 1111:1110 0011: mod mmreg r/m
PEXTRW—Extract Word	
mmreg to reg32, imm8	0000 1111:1100 0101:11 r32 mmreg: imm8
PINSRW—Insert Word	
reg32 to mmreg, imm8	0000 1111:1100 0100:11 mmreg r32: imm8
m16 to mmreg, imm8	0000 1111:1100 0100: mod mmreg r/m: imm8
PMAXSW—Maximum of Packed Signed Word Integers	
mmreg2 to mmreg1	0000 1111:1110 1110:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1110 1110: mod mmreg r/m
PMAXUB—Maximum of Packed Unsigned Byte Integers	
mmreg2 to mmreg1	0000 1111:1101 1110:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1101 1110: mod mmreg r/m
PMINSW—Minimum of Packed Signed Word Integers	
mmreg2 to mmreg1	0000 1111:1110 1010:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1110 1010: mod mmreg r/m
PMINUB—Minimum of Packed Unsigned Byte Integers	
mmreg2 to mmreg1	0000 1111:1101 1010:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1101 1010: mod mmreg r/m
PMOVMSKB—Move Byte Mask To Integer	
mmreg to reg32	0000 1111:1101 0111:11 r32 mmreg
PMULHUW—Multiply Packed Unsigned Integers and Store High Result	
mmreg2 to mmreg1	0000 1111:1110 0100:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1110 0100: mod mmreg r/m
PSADBW—Compute Sum of Absolute Differences	
mmreg2 to mmreg1	0000 1111:1111 0110:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1111 0110: mod mmreg r/m
PSHUFW—Shuffle Packed Words	
mmreg2 to mmreg1, imm8	0000 1111:0111 0000:11 mmreg1 mmreg2: imm8
mem to mmreg, imm8	0000 1111:0111 0000: mod mmreg r/m: imm8

Table B-24. Format and Encoding of SSE Cacheability & Memory Ordering Instructions

Instruction and Format	Encoding
MASKMOVQ—Store Selected Bytes of Quadword	
mmreg2 to mmreg1	0000 1111:1111 0111:11 mmreg1 mmreg2
MOVNTPS—Store Packed Single Precision Floating-Point Values Using Non-Temporal Hint	
xmmreg to mem	0000 1111:0010 1011: mod xmmreg r/m
MOVNTQ—Store Quadword Using Non-Temporal Hint	
mmreg to mem	0000 1111:1110 0111: mod mmreg r/m
PREFETCHT0—Prefetch Temporal to All Cache Levels	0000 1111:0001 1000:mod[A] 001 mem
PREFETCHT1—Prefetch Temporal to First Level Cache	0000 1111:0001 1000:mod[A] 010 mem
PREFETCHT2—Prefetch Temporal to Second Level Cache	0000 1111:0001 1000:mod[A] 011 mem
PREFETCHNTA—Prefetch Non-Temporal to All Cache Levels	0000 1111:0001 1000:mod[A] 000 mem
SFENCE—Store Fence	0000 1111:1010 1110:11 111 000

B.9 SSE2 INSTRUCTION FORMATS AND ENCODINGS

The SSE2 instructions use the ModR/M format and are preceded by the 0FH prefix byte. In general, operations are not duplicated to provide two directions (that is, separate load and store variants).

The following three tables show the formats and encodings for the SSE2 SIMD floating-point, SIMD integer, and cacheability instructions, respectively. Some SSE2 instructions require a mandatory prefix (66H, F2H, F3H) as part of the two-byte opcode. These prefixes are included in the tables.

B.9.1 Granularity Field (gg)

The granularity field (gg) indicates the size of the packed operands that the instruction is operating on. When this field is used, it is located in bits 1 and 0 of the second opcode byte. Table B-25 shows the encoding of this gg field.

Table B-25. Encoding of Granularity of Data Field (gg)

gg	Granularity of Data
00	Packed Bytes
01	Packed Words
10	Packed Doublewords
11	Quadword

Table B-26. Formats and Encodings of SSE2 Floating-Point Instructions

Instruction and Format	Encoding
ADDPD—Add Packed Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 1000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1000: mod xmmreg r/m
ADDSD—Add Scalar Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	1111 0010:0000 1111:0101 1000:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 1000: mod xmmreg r/m
ANDNPD—Bitwise Logical AND NOT of Packed Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 0101: mod xmmreg r/m
ANDPD—Bitwise Logical AND of Packed Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 0100: mod xmmreg r/m
CMPPD—Compare Packed Double Precision Floating-Point Values	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:1100 0010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:1100 0010: mod xmmreg r/m: imm8
CMPSD—Compare Scalar Double Precision Floating-Point Values	
xmmreg2 to xmmreg1, imm8	1111 0010:0000 1111:1100 0010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	11110 010:0000 1111:1100 0010: mod xmmreg r/m: imm8
COMISD—Compare Scalar Ordered Double Precision Floating-Point Values and Set EFLAGS	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0010 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0010 1111: mod xmmreg r/m
CVTPI2PD—Convert Packed Doubleword Integers to Packed Double Precision Floating-Point Values	
mmreg to xmmreg	0110 0110:0000 1111:0010 1010:11 xmmreg1 mmreg1
mem to xmmreg	0110 0110:0000 1111:0010 1010: mod xmmreg r/m
CVTPD2PI—Convert Packed Double Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to mmreg	0110 0110:0000 1111:0010 1101:11 mmreg1 xmmreg1
mem to mmreg	0110 0110:0000 1111:0010 1101: mod mmreg r/m
CVTSI2SD—Convert Signed Integer to Scalar Double Precision Floating-Point Value	
r32 to xmmreg1	1111 0010:0000 1111:0010 1010:11 xmmreg r32
mem to xmmreg	1111 0010:0000 1111:0010 1010: mod xmmreg r/m
CVTSD2SI—Convert Scalar Double Precision Floating-Point Value to Signed Integer	
xmmreg to r32	1111 0010:0000 1111:0010 1101:11 r32 xmmreg
mem to r32	1111 0010:0000 1111:0010 1101: mod r32 r/m
CVTTPD2PI—Convert with Truncation Packed Double Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to mmreg	0110 0110:0000 1111:0010 1100:11 mmreg xmmreg
mem to mmreg	0110 0110:0000 1111:0010 1100: mod mmreg r/m
CVTTSD2SI—Convert with Truncation Scalar Double Precision Floating-Point Value to Signed Integer	
xmmreg to r32	1111 0010:0000 1111:0010 1100:11 r32 xmmreg

Table B-26. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
mem to r32	1111 0010:0000 1111:0010 1100: mod r32 r/m
CVTPD2PS—Covert Packed Double Precision Floating-Point Values to Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 1010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1010: mod xmmreg r/m
CVTPS2PD—Covert Packed Single Precision Floating-Point Values to Packed Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 1010:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1010: mod xmmreg r/m
CVTSD2SS—Covert Scalar Double Precision Floating-Point Value to Scalar Single Precision Floating-Point Value	
xmmreg2 to xmmreg1	1111 0010:0000 1111:0101 1010:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 1010: mod xmmreg r/m
CVTSS2SD—Covert Scalar Single Precision Floating-Point Value to Scalar Double Precision Floating-Point Value	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 1010:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:00001 111:0101 1010: mod xmmreg r/m
CVTPD2DQ—Convert Packed Double Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg2 to xmmreg1	1111 0010:0000 1111:1110 0110:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:1110 0110: mod xmmreg r/m
CVTTPD2DQ—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 0110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1110 0110: mod xmmreg r/m
CVTDQ2PD—Convert Packed Doubleword Integers to Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	1111 0011:0000 1111:1110 0110:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:1110 0110: mod xmmreg r/m
CVTPS2DQ—Convert Packed Single Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 1011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1011: mod xmmreg r/m
CVTTPS2DQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 1011:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 1011: mod xmmreg r/m
CVTDQ2PS—Convert Packed Doubleword Integers to Packed Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 1011:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1011: mod xmmreg r/m
DIVPD—Divide Packed Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1110: mod xmmreg r/m
DIVSD—Divide Scalar Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	1111 0010:0000 1111:0101 1110:11 xmmreg1 xmmreg2

Table B-26. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
mem to xmmreg	1111 0010:0000 1111:0101 1110: mod xmmreg r/m
MAXPD—Return Maximum Packed Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1111: mod xmmreg r/m
MAXSD—Return Maximum Scalar Double Precision Floating-Point Value	
xmmreg2 to xmmreg1	1111 0010:0000 1111:0101 1111:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 1111: mod xmmreg r/m
MINPD—Return Minimum Packed Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1101: mod xmmreg r/m
MINSD—Return Minimum Scalar Double Precision Floating-Point Value	
xmmreg2 to xmmreg1	1111 0010:0000 1111:0101 1101:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 1101: mod xmmreg r/m
MOVAPD—Move Aligned Packed Double Precision Floating-Point Values	
xmmreg1 to xmmreg2	0110 0110:0000 1111:0010 1001:11 xmmreg2 xmmreg1
xmmreg1 to mem	0110 0110:0000 1111:0010 1001: mod xmmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0010 1000:11 xmmreg1 xmmreg2
mem to xmmreg1	0110 0110:0000 1111:0010 1000: mod xmmreg r/m
MOVHPD—Move High Packed Double Precision Floating-Point Values	
xmmreg to mem	0110 0110:0000 1111:0001 0111: mod xmmreg r/m
mem to xmmreg	0110 0110:0000 1111:0001 0110: mod xmmreg r/m
MOVLPD—Move Low Packed Double Precision Floating-Point Values	
xmmreg to mem	0110 0110:0000 1111:0001 0011: mod xmmreg r/m
mem to xmmreg	0110 0110:0000 1111:0001 0010: mod xmmreg r/m
MOVMSKPD—Extract Packed Double Precision Floating-Point Sign Mask	
xmmreg to r32	0110 0110:0000 1111:0101 0000:11 r32 xmmreg
MOVSD—Move Scalar Double Precision Floating-Point Values	
xmmreg1 to xmmreg2	1111 0010:0000 1111:0001 0001:11 xmmreg2 xmmreg1
xmmreg1 to mem	1111 0010:0000 1111:0001 0001: mod xmmreg r/m
xmmreg2 to xmmreg1	1111 0010:0000 1111:0001 0000:11 xmmreg1 xmmreg2
mem to xmmreg1	1111 0010:0000 1111:0001 0000: mod xmmreg r/m
MOVUPD—Move Unaligned Packed Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0001 0001:11 xmmreg2 xmmreg1
mem to xmmreg1	0110 0110:0000 1111:0001 0001: mod xmmreg r/m
xmmreg1 to xmmreg2	0110 0110:0000 1111:0001 0000:11 xmmreg1 xmmreg2
xmmreg1 to mem	0110 0110:0000 1111:0001 0000: mod xmmreg r/m
MULPD—Multiply Packed Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 1001:11 xmmreg1 xmmreg2

Table B-26. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
mem to xmmreg	0110 0110:0000 1111:0101 1001: mod xmmreg r/m
MULSD—Multiply Scalar Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	1111 0010:00001111:01011001:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:00001111:01011001: mod xmmreg r/m
ORPD—Bitwise Logical OR of Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 0110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 0110: mod xmmreg r/m
SHUFPD—Shuffle Packed Double Precision Floating-Point Values	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:1100 0110:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:1100 0110: mod xmmreg r/m: imm8
SQRTPD—Compute Square Roots of Packed Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 0001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 0001: mod xmmreg r/m
SQRTSD—Compute Square Root of Scalar Double Precision Floating-Point Value	
xmmreg2 to xmmreg1	1111 0010:0000 1111:0101 0001:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 0001: mod xmmreg r/m
SUBPD—Subtract Packed Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 1100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1100: mod xmmreg r/m
SUBSD—Subtract Scalar Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	1111 0010:0000 1111:0101 1100:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 1100: mod xmmreg r/m
UCOMISD—Unordered Compare Scalar Ordered Double Precision Floating-Point Values and Set EFLAGS	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0010 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0010 1110: mod xmmreg r/m
UNPCKHPD—Unpack and Interleave High Packed Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0001 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0001 0101: mod xmmreg r/m
UNPCKLPD—Unpack and Interleave Low Packed Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0001 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0001 0100: mod xmmreg r/m
XORPD—Bitwise Logical OR of Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 0111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 0111: mod xmmreg r/m

Table B-27. Formats and Encodings of SSE2 Integer Instructions

Instruction and Format	Encoding
MOVD—Move Doubleword	
reg to xmmreg	0110 0110:0000 1111:0110 1110: 11 xmmreg reg
reg from xmmreg	0110 0110:0000 1111:0111 1110: 11 xmmreg reg
mem to xmmreg	0110 0110:0000 1111:0110 1110: mod xmmreg r/m
mem from xmmreg	0110 0110:0000 1111:0111 1110: mod xmmreg r/m
MOVDQA—Move Aligned Double Quadword	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 1111:11 xmmreg1 xmmreg2
xmmreg2 from xmmreg1	0110 0110:0000 1111:0111 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0110 1111: mod xmmreg r/m
mem from xmmreg	0110 0110:0000 1111:0111 1111: mod xmmreg r/m
MOVDQU—Move Unaligned Double Quadword	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0110 1111:11 xmmreg1 xmmreg2
xmmreg2 from xmmreg1	1111 0011:0000 1111:0111 1111:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0110 1111: mod xmmreg r/m
mem from xmmreg	1111 0011:0000 1111:0111 1111: mod xmmreg r/m
MOVQ2DQ—Move Quadword from MMX to XMM Register	
mmreg to xmmreg	1111 0011:0000 1111:1101 0110:11 mmreg1 mmreg2
MOVDQ2Q—Move Quadword from XMM to MMX Register	
xmmreg to mmreg	1111 0010:0000 1111:1101 0110:11 mmreg1 mmreg2
MOVQ—Move Quadword	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0111 1110: 11 xmmreg1 xmmreg2
xmmreg2 from xmmreg1	0110 0110:0000 1111:1101 0110: 11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0111 1110: mod xmmreg r/m
mem from xmmreg	0110 0110:0000 1111:1101 0110: mod xmmreg r/m
PACKSSDW[1]—Pack Dword To Word Data (signed with saturation)	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 1011: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:0110 1011: mod xmmreg r/m
PACKSSWB—Pack Word To Byte Data (signed with saturation)	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 0011: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:0110 0011: mod xmmreg r/m
PACKUSWB—Pack Word To Byte Data (unsigned with saturation)	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 0111: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:0110 0111: mod xmmreg r/m
PADDQ—Add Packed Quadword Integers	
mmreg2 to mmreg1	0000 1111:1101 0100:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1101 0100: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:1101 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1101 0100: mod xmmreg r/m

Table B-27. Formats and Encodings of SSE2 Integer Instructions (Contd.)

Instruction and Format	Encoding
PADD—Add With Wrap-around	
xmmreg2 to xmmreg1	0110 0110:0000 1111: 1111 11gg: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111: 1111 11gg: mod xmmreg r/m
PADDS—Add Signed With Saturation	
xmmreg2 to xmmreg1	0110 0110:0000 1111: 1110 11gg: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111: 1110 11gg: mod xmmreg r/m
PADDUS—Add Unsigned With Saturation	
xmmreg2 to xmmreg1	0110 0110:0000 1111: 1101 11gg: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111: 1101 11gg: mod xmmreg r/m
PAND—Bitwise And	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1101 1011: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1101 1011: mod xmmreg r/m
PANDN—Bitwise AndNot	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1101 1111: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1101 1111: mod xmmreg r/m
PAVGB—Average Packed Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:11100 000:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11100000 mod xmmreg r/m
PAVGW—Average Packed Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 0011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1110 0011 mod xmmreg r/m
PCMPEQ—Packed Compare For Equality	
xmmreg1 with xmmreg2	0110 0110:0000 1111:0111 01gg: 11 xmmreg1 xmmreg2
xmmreg with memory	0110 0110:0000 1111:0111 01gg: mod xmmreg r/m
PCMPGT—Packed Compare Greater (signed)	
xmmreg1 with xmmreg2	0110 0110:0000 1111:0110 01gg: 11 xmmreg1 xmmreg2
xmmreg with memory	0110 0110:0000 1111:0110 01gg: mod xmmreg r/m
PEXTRW—Extract Word	
xmmreg to reg32, imm8	0110 0110:0000 1111:1100 0101:11 r32 xmmreg: imm8
PINSRW—Insert Word	
reg32 to xmmreg, imm8	0110 0110:0000 1111:1100 0100:11 xmmreg r32: imm8
m16 to xmmreg, imm8	0110 0110:0000 1111:1100 0100: mod xmmreg r/m: imm8
PMADDWD—Packed Multiply Add	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1111 0101: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1111 0101: mod xmmreg r/m
PMAXSW—Maximum of Packed Signed Word Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 1110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11101110: mod xmmreg r/m

Table B-27. Formats and Encodings of SSE2 Integer Instructions (Contd.)

Instruction and Format	Encoding
PMAXUB—Maximum of Packed Unsigned Byte Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1101 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1101 1110: mod xmmreg r/m
PMINSW—Minimum of Packed Signed Word Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 1010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1110 1010: mod xmmreg r/m
PMINUB—Minimum of Packed Unsigned Byte Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1101 1010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1101 1010 mod xmmreg r/m
PMOVMSKB—Move Byte Mask To Integer	
xmmreg to reg32	0110 0110:0000 1111:1101 0111:11 r32 xmmreg
PMULHUW—Packed multiplication, store high word (unsigned)	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 0100: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1110 0100: mod xmmreg r/m
PMULHW—Packed Multiplication, store high word	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 0101: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1110 0101: mod xmmreg r/m
PMULLW—Packed Multiplication, store low word	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1101 0101: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1101 0101: mod xmmreg r/m
PMULUDQ—Multiply Packed Unsigned Doubleword Integers	
mmreg2 to mmreg1	0000 1111:1111 0100:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1111 0100: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:00001111:1111 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:00001111:1111 0100: mod xmmreg r/m
POR—Bitwise Or	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 1011: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1110 1011: mod xmmreg r/m
PSADBW—Compute Sum of Absolute Differences	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1111 0110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1111 0110: mod xmmreg r/m
PSHUFLW—Shuffle Packed Low Words	
xmmreg2 to xmmreg1, imm8	1111 0010:0000 1111:0111 0000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	1111 0010:0000 1111:0111 0000:11 mod xmmreg r/m: imm8
PSHUFHW—Shuffle Packed High Words	
xmmreg2 to xmmreg1, imm8	1111 0011:0000 1111:0111 0000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	1111 0011:0000 1111:0111 0000: mod xmmreg r/m: imm8
PSHUFD—Shuffle Packed Doublewords	

Table B-27. Formats and Encodings of SSE2 Integer Instructions (Contd.)

Instruction and Format	Encoding
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0111 0000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0111 0000: mod xmmreg r/m: imm8
PSLLDQ—Shift Double Quadword Left Logical	
xmmreg, imm8	0110 0110:0000 1111:0111 0011:11 111 xmmreg: imm8
PSLL—Packed Shift Left Logical	
xmmreg1 by xmmreg2	0110 0110:0000 1111:1111 00gg: 11 xmmreg1 xmmreg2
xmmreg by memory	0110 0110:0000 1111:1111 00gg: mod xmmreg r/m
xmmreg by immediate	0110 0110:0000 1111:0111 00gg: 11 110 xmmreg: imm8
PSRA—Packed Shift Right Arithmetic	
xmmreg1 by xmmreg2	0110 0110:0000 1111:1110 00gg: 11 xmmreg1 xmmreg2
xmmreg by memory	0110 0110:0000 1111:1110 00gg: mod xmmreg r/m
xmmreg by immediate	0110 0110:0000 1111:0111 00gg: 11 100 xmmreg: imm8
PSRLDQ—Shift Double Quadword Right Logical	
xmmreg, imm8	0110 0110:00001111:01110011:11 011 xmmreg: imm8
PSRL—Packed Shift Right Logical	
xmmreg1 by xmmreg2	0110 0110:0000 1111:1101 00gg: 11 xmmreg1 xmmreg2
xmmreg by memory	0110 0110:0000 1111:1101 00gg: mod xmmreg r/m
xmmreg by immediate	0110 0110:0000 1111:0111 00gg: 11 010 xmmreg: imm8
PSUBQ—Subtract Packed Quadword Integers	
mmreg2 to mmreg1	0000 1111:11111 011:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1111 1011: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:1111 1011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1111 1011: mod xmmreg r/m
PSUB—Subtract With Wrap-around	
xmmreg2 from xmmreg1	0110 0110:0000 1111:1111 10gg: 11 xmmreg1 xmmreg2
memory from xmmreg	0110 0110:0000 1111:1111 10gg: mod xmmreg r/m
PSUBS—Subtract Signed With Saturation	
xmmreg2 from xmmreg1	0110 0110:0000 1111:1110 10gg: 11 xmmreg1 xmmreg2
memory from xmmreg	0110 0110:0000 1111:1110 10gg: mod xmmreg r/m
PSUBUS—Subtract Unsigned With Saturation	
xmmreg2 from xmmreg1	0000 1111:1101 10gg: 11 xmmreg1 xmmreg2
memory from xmmreg	0000 1111:1101 10gg: mod xmmreg r/m
PUNPCKH—Unpack High Data To Next Larger Type	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 10gg:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0110 10gg: mod xmmreg r/m
PUNPCKHQDQ—Unpack High Data	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0110 1101: mod xmmreg r/m

Table B-27. Formats and Encodings of SSE2 Integer Instructions (Contd.)

Instruction and Format	Encoding
PUNPCKL—Unpack Low Data To Next Larger Type	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 00gg:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0110 00gg: mod xmmreg r/m
PUNPCKLQDQ—Unpack Low Data	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 1100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0110 1100: mod xmmreg r/m
PXOR—Bitwise Xor	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 1111: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1110 1111: mod xmmreg r/m

Table B-28. Format and Encoding of SSE2 Cacheability Instructions

Instruction and Format	Encoding
MASKMOVDQU—Store Selected Bytes of Double Quadword	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1111 0111:11 xmmreg1 xmmreg2
CLFLUSH—Flush Cache Line	
mem	0000 1111:1010 1110: mod 111 r/m
MOVNTPD—Store Packed Double Precision Floating-Point Values Using Non-Temporal Hint	
xmmreg to mem	0110 0110:0000 1111:0010 1011: mod xmmreg r/m
MOVNTDQ—Store Double Quadword Using Non-Temporal Hint	
xmmreg to mem	0110 0110:0000 1111:1110 0111: mod xmmreg r/m
MOVNTI—Store Doubleword Using Non-Temporal Hint	
reg to mem	0000 1111:1100 0011: mod reg r/m
PAUSE—Spin Loop Hint	1111 0011:1001 0000
LFENCE—Load Fence	0000 1111:1010 1110: 11 101 000
MFENCE—Memory Fence	0000 1111:1010 1110: 11 110 000

B.10 SSE3 FORMATS AND ENCODINGS TABLE

The tables in this section provide SSE3 formats and encodings. Some SSE3 instructions require a mandatory prefix (66H, F2H, F3H) as part of the two-byte opcode. These prefixes are included in the tables.

When in IA-32e mode, use of the REX.R prefix permits instructions that use general purpose and XMM registers to access additional registers. Some instructions require the REX.W prefix to promote the instruction to 64-bit operation. Instructions that require the REX.W prefix are listed (with their opcodes) in Section B.13.

Table B-29. Formats and Encodings of SSE3 Floating-Point Instructions

Instruction and Format	Encoding
ADDSUBPD—Add /Sub packed DP FP numbers from XMM2/Mem to XMM1	
xmmreg2 to xmmreg1	01100110:00001111:11010000:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11010000: mod xmmreg r/m
ADDSUBPS—Add /Sub packed SP FP numbers from XMM2/Mem to XMM1	
xmmreg2 to xmmreg1	11110010:00001111:11010000:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:11010000: mod xmmreg r/m
HADDPD—Add horizontally packed DP FP numbers XMM2/Mem to XMM1	
xmmreg2 to xmmreg1	01100110:00001111:01111100:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01111100: mod xmmreg r/m
HADDPS—Add horizontally packed SP FP numbers XMM2/Mem to XMM1	
xmmreg2 to xmmreg1	11110010:00001111:01111100:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01111100: mod xmmreg r/m
HSUBPD—Sub horizontally packed DP FP numbers XMM2/Mem to XMM1	
xmmreg2 to xmmreg1	01100110:00001111:01111101:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01111101: mod xmmreg r/m
HSUBPS—Sub horizontally packed SP FP numbers XMM2/Mem to XMM1	
xmmreg2 to xmmreg1	11110010:00001111:01111101:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01111101: mod xmmreg r/m

Table B-30. Formats and Encodings for SSE3 Event Management Instructions

Instruction and Format	Encoding
MONITOR—Set up a linear address range to be monitored by hardware	
eax, ecx, edx	0000 1111 : 0000 0001:11 001 000
MWAIT—Wait until write-back store performed within the range specified by the instruction MONITOR	
eax, ecx	0000 1111 : 0000 0001:11 001 001

Table B-31. Formats and Encodings for SSE3 Integer and Move Instructions

Instruction and Format	Encoding
FISTTP—Store ST in int16 (chop) and pop	
m16int	11011 111 : mod[A] 001 r/m
FISTTP—Store ST in int32 (chop) and pop	
m32int	11011 011 : mod[A] 001 r/m
FISTTP—Store ST in int64 (chop) and pop	

Table B-31. Formats and Encodings for SSE3 Integer and Move Instructions (Contd.)

Instruction and Format	Encoding
m64int	11011 101 : mod[A] 001 r/m
LDDQU—Load unaligned integer 128-bit	
xmm, m128	11110010:00001111:11110000: mod[A] xmmreg r/m
MOVDDUP—Move 64 bits representing one DP data from XMM2/Mem to XMM1 and duplicate	
xmmreg2 to xmmreg1	11110010:00001111:00010010:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:00010010: mod xmmreg r/m
MOVSHDUP—Move 128 bits representing 4 SP data from XMM2/Mem to XMM1 and duplicate high	
xmmreg2 to xmmreg1	11110011:00001111:00010110:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:00010110: mod xmmreg r/m
MOVSLDUP—Move 128 bits representing 4 SP data from XMM2/Mem to XMM1 and duplicate low	
xmmreg2 to xmmreg1	11110011:00001111:00010010:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:00010010: mod xmmreg r/m

B.11 SSSE3 FORMATS AND ENCODING TABLE

The tables in this section provide SSSE3 formats and encodings. Some SSSE3 instructions require a mandatory prefix (66H) as part of the three-byte opcode. These prefixes are included in the table below.

Table B-32. Formats and Encodings for SSSE3 Instructions

Instruction and Format	Encoding
PABSB—Packed Absolute Value Bytes	
mmreg2 to mmreg1	0000 1111:0011 1000: 0001 1100:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0001 1100: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0001 1100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0001 1100: mod xmmreg r/m
PABSD—Packed Absolute Value Double Words	
mmreg2 to mmreg1	0000 1111:0011 1000: 0001 1110:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0001 1110: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0001 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0001 1110: mod xmmreg r/m
PABSW—Packed Absolute Value Words	
mmreg2 to mmreg1	0000 1111:0011 1000: 0001 1101:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0001 1101: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0001 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0001 1101: mod xmmreg r/m
PALIGNR—Packed Align Right	
mmreg2 to mmreg1, imm8	0000 1111:0011 1010: 0000 1111:11 mmreg1 mmreg2: imm8

Table B-32. Formats and Encodings for SSSE3 Instructions (Contd.)

Instruction and Format	Encoding
mem to mmreg, imm8	0000 1111:0011 1010: 0000 1111: mod mmreg r/m: imm8
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0000 1111:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0000 1111: mod xmmreg r/m: imm8
PHADDD—Packed Horizontal Add Double Words	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 0010:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0010: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 0010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0010: mod xmmreg r/m
PHADDSW—Packed Horizontal Add and Saturate	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 0011:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0011: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 0011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0011: mod xmmreg r/m
PHADDW—Packed Horizontal Add Words	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 0001:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0001: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 0001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0001: mod xmmreg r/m
PHSUBD—Packed Horizontal Subtract Double Words	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 0110:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0110: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 0110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0110: mod xmmreg r/m
PHSUBSW—Packed Horizontal Subtract and Saturate	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 0111:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0111: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 0111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0111: mod xmmreg r/m
PHSUBW—Packed Horizontal Subtract Words	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 0101:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0101: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 0101:11 xmmreg1 xmmreg2

Table B-32. Formats and Encodings for SSSE3 Instructions (Contd.)

Instruction and Format	Encoding
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0101: mod xmmreg r/m
PMADDUBSW—Multiply and Add Packed Signed and Unsigned Bytes	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 0100:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0100: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0100: mod xmmreg r/m
PMULHRSW—Packed Multiply HIgn with Round and Scale	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 1011:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 1011: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 1011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 1011: mod xmmreg r/m
PSHUFB—Packed Shuffle Bytes	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 0000:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0000: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 0000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0000: mod xmmreg r/m
PSIGNB—Packed Sign Bytes	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 1000:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 1000: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 1000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 1000: mod xmmreg r/m
PSIGND—Packed Sign Double Words	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 1010:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 1010: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 1010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 1010: mod xmmreg r/m
PSIGNW—Packed Sign Words	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 1001:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 1001: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 1001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 1001: mod xmmreg r/m

B.12 AESNI AND PCLMULQDQ INSTRUCTION FORMATS AND ENCODINGS

Table B-33 shows the formats and encodings for AESNI and PCLMULQDQ instructions.

Table B-33. Formats and Encodings of AESNI and PCLMULQDQ Instructions

Instruction and Format	Encoding
AESDEC—Perform One Round of an AES Decryption Flow	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000:1101 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000:1101 1110: mod xmmreg r/m
AESDECLAST—Perform Last Round of an AES Decryption Flow	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000:1101 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000:1101 1111: mod xmmreg r/m
AESENC—Perform One Round of an AES Encryption Flow	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000:1101 1100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000:1101 1100: mod xmmreg r/m
AESENCLAST—Perform Last Round of an AES Encryption Flow	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000:1101 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000:1101 1101: mod xmmreg r/m
AESIMC—Perform the AES InvMixColumn Transformation	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000:1101 1011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000:1101 1011: mod xmmreg r/m
AESKEYGENASSIST—AES Round Key Generation Assist	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010:1101 1111:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010:1101 1111: mod xmmreg r/m: imm8
PCLMULQDQ—Carry-Less Multiplication Quadword	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010:0100 0100:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010:0100 0100: mod xmmreg r/m: imm8

B.13 SPECIAL ENCODINGS FOR 64-BIT MODE

The following Pentium, P6, MMX, SSE, SSE2, SSE3 instructions are promoted to 64-bit operation in IA-32e mode by using REX.W. However, these entries are special cases that do not follow the general rules (specified in Section B.4).

Table B-34. Special Case Instructions Promoted Using REX.W

Instruction and Format	Encoding
CMOVcc—Conditional Move	

Table B-34. Special Case Instructions Promoted Using REX.W (Contd.)

Instruction and Format	Encoding
register2 to register1	0100 0R0B 0000 1111: 0100 tttn : 11 reg1 reg2
qwordregister2 to qwordregister1	0100 1R0B 0000 1111: 0100 tttn : 11 qwordreg1 qwordreg2
memory to register	0100 0RXB 0000 1111 : 0100 tttn : mod reg r/m
memory64 to qwordregister	0100 1RXB 0000 1111 : 0100 tttn : mod qwordreg r/m
CVTSD2SI—Convert Scalar Double Precision Floating-Point Value to Signed Integer	
xmmreg to r32	0100 0R0B 1111 0010:0000 1111:0010 1101:11 r32 xmmreg
xmmreg to r64	0100 1R0B 1111 0010:0000 1111:0010 1101:11 r64 xmmreg
mem64 to r32	0100 0R0XB 1111 0010:0000 1111:0010 1101: mod r32 r/m
mem64 to r64	0100 1RXB 1111 0010:0000 1111:0010 1101: mod r64 r/m
CVTSI2SS—Convert Signed Integer to Scalar Single Precision Floating-Point Value	
r32 to xmmreg1	0100 0R0B 1111 0011:0000 1111:0010 1010:11 xmmreg r32
r64 to xmmreg1	0100 1R0B 1111 0011:0000 1111:0010 1010:11 xmmreg r64
mem to xmmreg	0100 0RXB 1111 0011:0000 1111:0010 1010: mod xmmreg r/m
mem64 to xmmreg	0100 1RXB 1111 0011:0000 1111:0010 1010: mod xmmreg r/m
CVTSI2SD—Convert Signed Integer to Scalar Double Precision Floating-Point Value	
r32 to xmmreg1	0100 0R0B 1111 0010:0000 1111:0010 1010:11 xmmreg r32
r64 to xmmreg1	0100 1R0B 1111 0010:0000 1111:0010 1010:11 xmmreg r64
mem to xmmreg	0100 0RXB 1111 0010:0000 1111:00101 010: mod xmmreg r/m
mem64 to xmmreg	0100 1RXB 1111 0010:0000 1111:0010 1010: mod xmmreg r/m
CVTSS2SI—Convert Scalar Single Precision Floating-Point Value to Signed Integer	
xmmreg to r32	0100 0R0B 1111 0011:0000 1111:0010 1101:11 r32 xmmreg
xmmreg to r64	0100 1R0B 1111 0011:0000 1111:0010 1101:11 r64 xmmreg
mem to r32	0100 0RXB 11110011:00001111:00101101: mod r32 r/m
mem32 to r64	0100 1RXB 1111 0011:0000 1111:0010 1101: mod r64 r/m
CVTTSD2SI—Convert with Truncation Scalar Double Precision Floating-Point Value to Signed Integer	
xmmreg to r32	0100 0R0B 11110010:00001111:00101100:11 r32 xmmreg
xmmreg to r64	0100 1R0B 1111 0010:0000 1111:0010 1100:11 r64 xmmreg
mem64 to r32	0100 0RXB 1111 0010:0000 1111:0010 1100: mod r32 r/m
mem64 to r64	0100 1RXB 1111 0010:0000 1111:0010 1100: mod r64 r/m

Table B-34. Special Case Instructions Promoted Using REX.W (Contd.)

Instruction and Format	Encoding
CVTTSS2SI—Convert with Truncation Scalar Single Precision Floating-Point Value to Signed Integer	
xmmreg to r32	0100 0R0B 1111 0011:0000 1111:0010 1100:11 r32 xmmreg1
xmmreg to r64	0100 1R0B 1111 0011:0000 1111:0010 1100:11 r64 xmmreg1
mem to r32	0100 0RXB 1111 0011:0000 1111:0010 1100: mod r32 r/m
mem32 to r64	0100 1RXB 1111 0011:0000 1111:0010 1100: mod r64 r/m
MOVD/MOVQ—Move doubleword	
reg to mmxreg	0100 0R0B 0000 1111:0110 1110: 11 mmxreg reg
qwordreg to mmxreg	0100 1R0B 0000 1111:0110 1110: 11 mmxreg qwordreg
reg from mmxreg	0100 0R0B 0000 1111:0111 1110: 11 mmxreg reg
qwordreg from mmxreg	0100 1R0B 0000 1111:0111 1110: 11 mmxreg qwordreg
mem to mmxreg	0100 0RXB 0000 1111:0110 1110: mod mmxreg r/m
mem64 to mmxreg	0100 1RXB 0000 1111:0110 1110: mod mmxreg r/m
mem from mmxreg	0100 0RXB 0000 1111:0111 1110: mod mmxreg r/m
mem64 from mmxreg	0100 1RXB 0000 1111:0111 1110: mod mmxreg r/m
mmxreg with memory	0100 0RXB 0000 1111:0110 01gg: mod mmxreg r/m
MOVMSKPS—Extract Packed Single Precision Floating-Point Sign Mask	
xmmreg to r32	0100 0R0B 0000 1111:0101 0000:11 r32 xmmreg
xmmreg to r64	0100 1R0B 00001111:01010000:11 r64 xmmreg
PEXTRW—Extract Word	
mmreg to reg32, imm8	0100 0R0B 0000 1111:1100 0101:11 r32 mmreg: imm8
mmreg to reg64, imm8	0100 1R0B 0000 1111:1100 0101:11 r64 mmreg: imm8
xmmreg to reg32, imm8	0100 0R0B 0110 0110 0000 1111:1100 0101:11 r32 xmmreg: imm8
xmmreg to reg64, imm8	0100 1R0B 0110 0110 0000 1111:1100 0101:11 r64 xmmreg: imm8
PINSRW—Insert Word	
reg32 to mmreg, imm8	0100 0R0B 0000 1111:1100 0100:11 mmreg r32: imm8
reg64 to mmreg, imm8	0100 1R0B 0000 1111:1100 0100:11 mmreg r64: imm8
m16 to mmreg, imm8	0100 0R0B 0000 1111:1100 0100 mod mmreg r/m: imm8
m16 to mmreg, imm8	0100 1RXB 0000 1111:11000100 mod mmreg r/m: imm8
reg32 to xmmreg, imm8	0100 0RXB 0110 0110 0000 1111:1100 0100:11 xmmreg r32: imm8
reg64 to xmmreg, imm8	0100 0RXB 0110 0110 0000 1111:1100 0100:11 xmmreg r64: imm8
m16 to xmmreg, imm8	0100 0RXB 0110 0110 0000 1111:1100 0100 mod xmmreg r/m: imm8
m16 to xmmreg, imm8	0100 1RXB 0110 0110 0000 1111:1100 0100 mod xmmreg r/m: imm8
PMOVMSKB—Move Byte Mask To Integer	

Table B-34. Special Case Instructions Promoted Using REX.W (Contd.)

Instruction and Format	Encoding
mmreg to reg32	0100 0RXB 0000 1111:1101 0111:11 r32 mmreg
mmreg to reg64	0100 1R0B 0000 1111:1101 0111:11 r64 mmreg
xmmreg to reg32	0100 0RXB 0110 0110 0000 1111:1101 0111:11 r32 mmreg
xmmreg to reg64	0110 0110 0000 1111:1101 0111:11 r64 xmmreg

B.14 SSE4.1 FORMATS AND ENCODING TABLE

The tables in this section provide SSE4.1 formats and encodings. Some SSE4.1 instructions require a mandatory prefix (66H, F2H, F3H) as part of the three-byte opcode. These prefixes are included in the tables.

In 64-bit mode, some instructions requires REX.W, the byte sequence of REX.W prefix in the opcode sequence is shown.

Table B-35. Encodings of SSE4.1 instructions

Instruction and Format	Encoding
BLENDPD — Blend Packed Double Precision Floats	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1010: 0000 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1010: 0000 1101: mod xmmreg r/m
BLENDPS — Blend Packed Single Precision Floats	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1010: 0000 1100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1010: 0000 1100: mod xmmreg r/m
BLENDVPD — Variable Blend Packed Double Precision Floats	
xmmreg2 to xmmreg1 <xmm0>	0110 0110:0000 1111:0011 1000: 0001 0101:11 xmmreg1 xmmreg2
mem to xmmreg <xmm0>	0110 0110:0000 1111:0011 1000: 0001 0101: mod xmmreg r/m
BLENDVPS — Variable Blend Packed Single Precision Floats	
xmmreg2 to xmmreg1 <xmm0>	0110 0110:0000 1111:0011 1000: 0001 0100:11 xmmreg1 xmmreg2
mem to xmmreg <xmm0>	0110 0110:0000 1111:0011 1000: 0001 0100: mod xmmreg r/m
DPPD — Packed Double Precision Dot Products	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0100 0001:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0100 0001: mod xmmreg r/m: imm8
DPPS — Packed Single Precision Dot Products	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0100 0000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0100 0000: mod xmmreg r/m: imm8
EXTRACTPS — Extract From Packed Single Precision Floats	
reg from xmmreg , imm8	0110 0110:0000 1111:0011 1010: 0001 0111:11 xmmreg reg: imm8

Table B-35. Encodings of SSE4.1 instructions

Instruction and Format	Encoding
mem from xmmreg , imm8	0110 0110:0000 1111:0011 1010: 0001 0111: mod xmmreg r/m: imm8
INSERTPS — Insert Into Packed Single Precision Floats	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0010 0001:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0010 0001: mod xmmreg r/m: imm8
MOVNTDQA — Load Double Quadword Non-temporal Aligned	
m128 to xmmreg	0110 0110:0000 1111:0011 1000: 0010 1010:11 r/m xmmreg2
MPSADBW — Multiple Packed Sums of Absolute Difference	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0100 0010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0100 0010: mod xmmreg r/m: imm8
PACKUSDW — Pack with Unsigned Saturation	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 1011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 1011: mod xmmreg r/m
PBLENDVB — Variable Blend Packed Bytes	
xmmreg2 to xmmreg1 <xmm0>	0110 0110:0000 1111:0011 1000: 0001 0000:11 xmmreg1 xmmreg2
mem to xmmreg <xmm0>	0110 0110:0000 1111:0011 1000: 0001 0000: mod xmmreg r/m
PBLENDW — Blend Packed Words	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0001 1110:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0000 1110: mod xmmreg r/m: imm8
PCMPEQQ — Compare Packed Qword Data of Equal	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 1001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 1001: mod xmmreg r/m
PEXTRB — Extract Byte	
reg from xmmreg , imm8	0110 0110:0000 1111:0011 1010: 0001 0100:11 xmmreg reg: imm8
xmmreg to mem, imm8	0110 0110:0000 1111:0011 1010: 0001 0100: mod xmmreg r/m: imm8
PEXTRD — Extract DWord	
reg from xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0001 0110:11 xmmreg reg: imm8
xmmreg to mem, imm8	0110 0110:0000 1111:0011 1010: 0001 0110: mod xmmreg r/m: imm8

Table B-35. Encodings of SSE4.1 instructions

Instruction and Format	Encoding
PEXTRQ — Extract QWord	
r64 from xmmreg, imm8	0110 0110:REX.W:0000 1111:0011 1010: 0001 0110:11 xmmreg reg: imm8
m64 from xmmreg, imm8	0110 0110:REX.W:0000 1111:0011 1010: 0001 0110: mod xmmreg r/m: imm8
PEXTRW — Extract Word	
reg from xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0001 0101:11 reg xmmreg: imm8
mem from xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0001 0101: mod xmmreg r/m: imm8
PHMINPOSUW — Packed Horizontal Word Minimum	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0100 0001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0100 0001: mod xmmreg r/m
PINSRB — Extract Byte	
reg to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0010 0000:11 xmmreg reg: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0010 0000: mod xmmreg r/m: imm8
PINSRD — Extract DWord	
reg to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0010 0010:11 xmmreg reg: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0010 0010: mod xmmreg r/m: imm8
PINSRQ — Extract QWord	
r64 to xmmreg, imm8	0110 0110:REX.W:0000 1111:0011 1010: 0010 0010:11 xmmreg reg: imm8
m64 to xmmreg, imm8	0110 0110:REX.W:0000 1111:0011 1010: 0010 0010: mod xmmreg r/m: imm8
PMAXSB — Maximum of Packed Signed Byte Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 1100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 1100: mod xmmreg r/m
PMAXSD — Maximum of Packed Signed Dword Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 1101: mod xmmreg r/m
PMAXUD — Maximum of Packed Unsigned Dword Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 1111: mod xmmreg r/m
PMAXUW — Maximum of Packed Unsigned Word Integers	

Table B-35. Encodings of SSE4.1 instructions

Instruction and Format	Encoding
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 1110: mod xmmreg r/m
PMINSB — Minimum of Packed Signed Byte Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 1000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 1000: mod xmmreg r/m
PMINSD — Minimum of Packed Signed Dword Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 1001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 1001: mod xmmreg r/m
PMINUD — Minimum of Packed Unsigned Dword Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 1011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 1011: mod xmmreg r/m
PMINUW — Minimum of Packed Unsigned Word Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 1010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 1010: mod xmmreg r/m
PMOVSXBD — Packed Move Sign Extend - Byte to Dword	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 0001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 0001: mod xmmreg r/m
PMOVSXBQ — Packed Move Sign Extend - Byte to Qword	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 0010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 0010: mod xmmreg r/m
PMOVSXBW — Packed Move Sign Extend - Byte to Word	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 0000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 0000: mod xmmreg r/m
PMOVSXWD — Packed Move Sign Extend - Word to Dword	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 0011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 0011: mod xmmreg r/m
PMOVSXWQ — Packed Move Sign Extend - Word to Qword	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 0100: mod xmmreg r/m
PMOVSXDQ — Packed Move Sign Extend - Dword to Qword	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 0101:11 xmmreg1 xmmreg2

Table B-35. Encodings of SSE4.1 instructions

Instruction and Format	Encoding
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 0101: mod xmmreg r/m
PMOVZXBD — Packed Move Zero Extend - Byte to Dword	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 0001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 0001: mod xmmreg r/m
PMOVZXBQ — Packed Move Zero Extend - Byte to Qword	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 0010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 0010: mod xmmreg r/m
PMOVZXBW — Packed Move Zero Extend - Byte to Word	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 0000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 0000: mod xmmreg r/m
PMOVZXWD — Packed Move Zero Extend - Word to Dword	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 0011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 0011: mod xmmreg r/m
PMOVZXWQ — Packed Move Zero Extend - Word to Qword	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 0100: mod xmmreg r/m
PMOVZXDQ — Packed Move Zero Extend - Dword to Qword	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 0101: mod xmmreg r/m
PMULDQ — Multiply Packed Signed Dword Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 1000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 1000: mod xmmreg r/m
PMULLD — Multiply Packed Signed Dword Integers, Store low Result	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0100 0000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0100 0000: mod xmmreg r/m
PTEST — Logical Compare	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0001 0111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0001 0111: mod xmmreg r/m
ROUNDPD — Round Packed Double Precision Values	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0000 1001:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0000 1001: mod xmmreg r/m: imm8

Table B-35. Encodings of SSE4.1 instructions

Instruction and Format	Encoding
ROUNDPS — Round Packed Single Precision Values	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0000 1000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0000 1000: mod xmmreg r/m: imm8
ROUNDSD — Round Scalar Double Precision Value	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0000 1011:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0000 1011: mod xmmreg r/m: imm8
ROUNDSS — Round Scalar Single Precision Value	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0000 1010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0000 1010: mod xmmreg r/m: imm8

B.15 SSE4.2 FORMATS AND ENCODING TABLE

The tables in this section provide SSE4.2 formats and encodings. Some SSE4.2 instructions require a mandatory prefix (66H, F2H, F3H) as part of the three-byte opcode. These prefixes are included in the tables. In 64-bit mode, some instructions requires REX.W, the byte sequence of REX.W prefix in the opcode sequence is shown.

Table B-36. Encodings of SSE4.2 instructions

Instruction and Format	Encoding
CRC32 — Accumulate CRC32	
reg2 to reg1	1111 0010:0000 1111:0011 1000: 1111 000w :11 reg1 reg2
mem to reg	1111 0010:0000 1111:0011 1000: 1111 000w : mod reg r/m
bytereg2 to reg1	1111 0010:0100 WR0B:0000 1111:0011 1000: 1111 0000 :11 reg1 bytereg2
m8 to reg	1111 0010:0100 WR0B:0000 1111:0011 1000: 1111 0000 : mod reg r/m
qwreg2 to qwreg1	1111 0010:0100 1R0B:0000 1111:0011 1000: 1111 0001 :11 qwreg1 qwreg2
mem64 to qwreg	1111 0010:0100 1R0B:0000 1111:0011 1000: 1111 0001 : mod qwreg r/m
PCMPESTRI— Packed Compare Explicit-Length Strings To Index	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0110 0001:11 xmmreg1 xmmreg2: imm8
mem to xmmreg	0110 0110:0000 1111:0011 1010: 0110 0001: mod xmmreg r/m
PCMPESTRM— Packed Compare Explicit-Length Strings To Mask	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0110 0000:11 xmmreg1 xmmreg2: imm8

Table B-36. Encodings of SSE4.2 instructions

Instruction and Format	Encoding
mem to xmmreg	0110 0110:0000 1111:0011 1010: 0110 0000: mod xmmreg r/m
PCMPISTRI— Packed Compare Implicit-Length String To Index	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0110 0011:11 xmmreg1 xmmreg2: imm8
mem to xmmreg	0110 0110:0000 1111:0011 1010: 0110 0011: mod xmmreg r/m
PCMPISTRM— Packed Compare Implicit-Length Strings To Mask	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0110 0010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg	0110 0110:0000 1111:0011 1010: 0110 0010: mod xmmreg r/m
PCMPGTQ— Packed Compare Greater Than	
xmmreg to xmmreg	0110 0110:0000 1111:0011 1000: 0011 0111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 0111: mod xmmreg r/m
POPCNT— Return Number of Bits Set to 1	
reg2 to reg1	1111 0011:0000 1111:1011 1000:11 reg1 reg2
mem to reg1	1111 0011:0000 1111:1011 1000:mod reg1 r/m
qwreg2 to qwreg1	1111 0011:0100 1R0B:0000 1111:1011 1000:11 reg1 reg2
mem64 to qwreg1	1111 0011:0100 1R0B:0000 1111:1011 1000:mod reg1 r/m

B.16 AVX FORMATS AND ENCODING TABLE

The tables in this section provide AVX formats and encodings. A mixed form of bit/hex/symbolic forms are used to express the various bytes:

The C4/C5 and opcode bytes are expressed in hex notation; the first and second payload byte of VEX, the modR/M byte is expressed in combination of bit/symbolic form. The first payload byte of C4 is expressed as combination of bits and hex form, with the hex value preceded by an underscore. The VEX bit field to encode upper register 8-15 uses 1's complement form, each of those bit field is expressed as lower case notation rxb, instead of RXB.

The hybrid bit-nibble-byte form is depicted below:

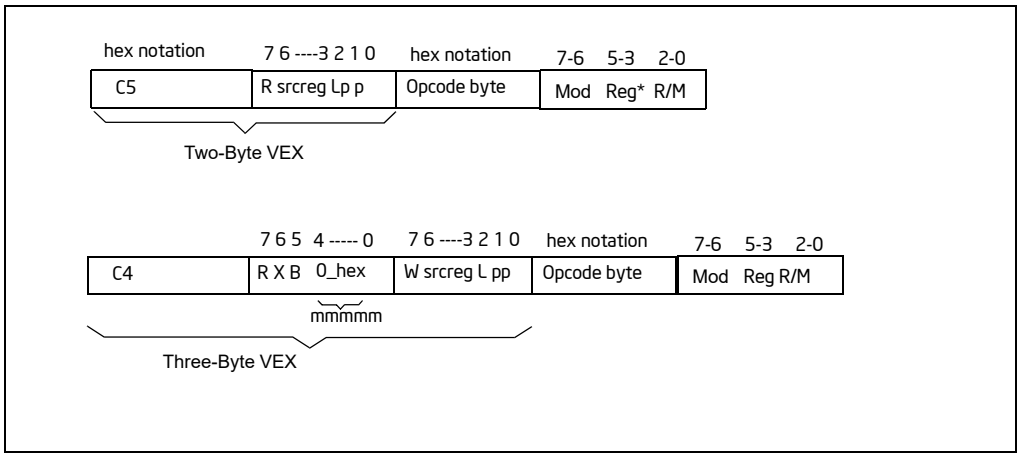


Figure B-2. Hybrid Notation of VEX-Encoded Key Instruction Bytes

Table B-37. Encodings of AVX Instructions

Instruction and Format	Encoding
VBLENDPD — Blend Packed Double Precision Floats	
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_3: w xmmreg2 001:0D:11 xmmreg1 xmmreg3: imm
xmmreg2 with mem to xmmreg1	C4: rxb0_3: w xmmreg2 001:0D:mod xmmreg1 r/m: imm
ymmreg2 with ymmreg3 into ymmreg1	C4: rxb0_3: w ymmreg2 101:0D:11 ymmreg1 ymmreg3: imm
ymmreg2 with mem to ymmreg1	C4: rxb0_3: w ymmreg2 101:0D:mod ymmreg1 r/m: imm
VBLENDPS — Blend Packed Single Precision Floats	
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_3: w xmmreg2 001:0C:11 xmmreg1 xmmreg3: imm
xmmreg2 with mem to xmmreg1	C4: rxb0_3: w xmmreg2 001:0C:mod xmmreg1 r/m: imm
ymmreg2 with ymmreg3 into ymmreg1	C4: rxb0_3: w ymmreg2 101:0C:11 ymmreg1 ymmreg3: imm
ymmreg2 with mem to ymmreg1	C4: rxb0_3: w ymmreg2 101:0C:mod ymmreg1 r/m: imm
VBLENDVPD — Variable Blend Packed Double Precision Floats	
xmmreg2 with xmmreg3 into xmmreg1 using xmmreg4 as mask	C4: rxb0_3: 0 xmmreg2 001:4B:11 xmmreg1 xmmreg3: xmmreg4
xmmreg2 with mem to xmmreg1 using xmmreg4 as mask	C4: rxb0_3: 0 xmmreg2 001:4B:mod xmmreg1 r/m: xmmreg4
ymmreg2 with ymmreg3 into ymmreg1 using ymmreg4 as mask	C4: rxb0_3: 0 ymmreg2 101:4B:11 ymmreg1 ymmreg3: ymmreg4
ymmreg2 with mem to ymmreg1 using ymmreg4 as mask	C4: rxb0_3: 0 ymmreg2 101:4B:mod ymmreg1 r/m: ymmreg4
VBLENDVPS — Variable Blend Packed Single Precision Floats	
xmmreg2 with xmmreg3 into xmmreg1 using xmmreg4 as mask	C4: rxb0_3: 0 xmmreg2 001:4A:11 xmmreg1 xmmreg3: xmmreg4
xmmreg2 with mem to xmmreg1 using xmmreg4 as mask	C4: rxb0_3: 0 xmmreg2 001:4A:mod xmmreg1 r/m: xmmreg4
ymmreg2 with ymmreg3 into ymmreg1 using ymmreg4 as mask	C4: rxb0_3: 0 ymmreg2 101:4A:11 ymmreg1 ymmreg3: ymmreg4
ymmreg2 with mem to ymmreg1 using ymmreg4 as mask	C4: rxb0_3: 0 ymmreg2 101:4A:mod ymmreg1 r/m: ymmreg4
VDPPD — Packed Double Precision Dot Products	
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_3: w xmmreg2 001:41:11 xmmreg1 xmmreg3: imm
xmmreg2 with mem to xmmreg1	C4: rxb0_3: w xmmreg2 001:41:mod xmmreg1 r/m: imm
VDPPS — Packed Single Precision Dot Products	
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_3: w xmmreg2 001:40:11 xmmreg1 xmmreg3: imm
xmmreg2 with mem to xmmreg1	C4: rxb0_3: w xmmreg2 001:40:mod xmmreg1 r/m: imm
ymmreg2 with ymmreg3 into ymmreg1	C4: rxb0_3: w ymmreg2 101:40:11 ymmreg1 ymmreg3: imm
ymmreg2 with mem to ymmreg1	C4: rxb0_3: w ymmreg2 101:40:mod ymmreg1 r/m: imm
VEXTRACTPS — Extract From Packed Single Precision Floats	
reg from xmmreg1 using imm	C4: rxb0_3: w_F 001:17:11 xmmreg1 reg: imm
mem from xmmreg1 using imm	C4: rxb0_3: w_F 001:17:mod xmmreg1 r/m: imm
VINSERTPS — Insert Into Packed Single Precision Floats	
use imm to merge xmmreg3 with xmmreg2 into xmmreg1	C4: rxb0_3: w xmmreg2 001:21:11 xmmreg1 xmmreg3: imm
use imm to merge mem with xmmreg2 into xmmreg1	C4: rxb0_3: w xmmreg2 001:21:mod xmmreg1 r/m: imm
VMOVNTDQA — Load Double Quadword Non-temporal Aligned	
m128 to xmmreg1	C4: rxb0_2: w_F 001:2A:11 xmmreg1 r/m

Instruction and Format	Encoding
VMPSADBW — Multiple Packed Sums of Absolute Difference	
xmmreg3 with xmmreg2 into xmmreg1	C4: rxb0_3: w xmmreg2 001:42:11 xmmreg1 xmmreg3: imm
m128 with xmmreg2 into xmmreg1	C4: rxb0_3: w xmmreg2 001:42:mod xmmreg1 r/m: imm
VPACKUSDW — Pack with Unsigned Saturation	
xmmreg3 and xmmreg2 to xmmreg1	C4: rxb0_2: w xmmreg2 001:2B:11 xmmreg1 xmmreg3: imm
m128 and xmmreg2 to xmmreg1	C4: rxb0_2: w xmmreg2 001:2B:mod xmmreg1 r/m: imm
VPBLENDVB — Variable Blend Packed Bytes	
xmmreg2 with xmmreg3 into xmmreg1 using xmmreg4 as mask	C4: rxb0_3: w xmmreg2 001:4C:11 xmmreg1 xmmreg3: xmmreg4
xmmreg2 with mem to xmmreg1 using xmmreg4 as mask	C4: rxb0_3: w xmmreg2 001:4C:mod xmmreg1 r/m: xmmreg4
VPBLENDW — Blend Packed Words	
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_3: w xmmreg2 001:0E:11 xmmreg1 xmmreg3: imm
xmmreg2 with mem to xmmreg1	C4: rxb0_3: w xmmreg2 001:0E:mod xmmreg1 r/m: imm
VPCMPEQQ — Compare Packed Qword Data of Equal	
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_2: w xmmreg2 001:29:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:29:mod xmmreg1 r/m:
VPEXTRB — Extract Byte	
reg from xmmreg1 using imm	C4: rxb0_3: 0_F 001:14:11 xmmreg1 reg: imm
mem from xmmreg1 using imm	C4: rxb0_3: 0_F 001:14:mod xmmreg1 r/m: imm
VPEXTRD — Extract DWord	
reg from xmmreg1 using imm	C4: rxb0_3: 0_F 001:16:11 xmmreg1 reg: imm
mem from xmmreg1 using imm	C4: rxb0_3: 0_F 001:16:mod xmmreg1 r/m: imm
VPEXTRQ — Extract QWord	
reg from xmmreg1 using imm	C4: rxb0_3: 1_F 001:16:11 xmmreg1 reg: imm
mem from xmmreg1 using imm	C4: rxb0_3: 1_F 001:16:mod xmmreg1 r/m: imm
VPEXTRW — Extract Word	
reg from xmmreg1 using imm	C4: rxb0_3: 0_F 001:15:11 xmmreg1 reg: imm
mem from xmmreg1 using imm	C4: rxb0_3: 0_F 001:15:mod xmmreg1 r/m: imm
VPHMINPOSUW — Packed Horizontal Word Minimum	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:41:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:41:mod xmmreg1 r/m
VPINSRB — Insert Byte	
reg with xmmreg2 to xmmreg1, imm8	C4: rxb0_3: 0 xmmreg2 001:20:11 xmmreg1 reg: imm
mem with xmmreg2 to xmmreg1, imm8	C4: rxb0_3: 0 xmmreg2 001:20:mod xmmreg1 r/m: imm
VPINSRD — Insert DWord	
reg with xmmreg2 to xmmreg1, imm8	C4: rxb0_3: 0 xmmreg2 001:22:11 xmmreg1 reg: imm
mem with xmmreg2 to xmmreg1, imm8	C4: rxb0_3: 0 xmmreg2 001:22:mod xmmreg1 r/m: imm
VPINSRQ — Insert QWord	
r64 with xmmreg2 to xmmreg1, imm8	C4: rxb0_3: 1 xmmreg2 001:22:11 xmmreg1 reg: imm

Instruction and Format	Encoding
m64 with xmmreg2 to xmmreg1, imm8	C4: rxb0_3: 1 xmmreg2 001:22:mod xmmreg1 r/m: imm
VPMAXSB — Maximum of Packed Signed Byte Integers	
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_2: w xmmreg2 001:3C:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:3C:mod xmmreg1 r/m
VPMAXSD — Maximum of Packed Signed Dword Integers	
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_2: w xmmreg2 001:3D:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:3D:mod xmmreg1 r/m
VPMAXUD — Maximum of Packed Unsigned Dword Integers	
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_2: w xmmreg2 001:3F:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:3F:mod xmmreg1 r/m
VPMAXUW — Maximum of Packed Unsigned Word Integers	
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_2: w xmmreg2 001:3E:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:3E:mod xmmreg1 r/m
VPMINSB — Minimum of Packed Signed Byte Integers	
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_2: w xmmreg2 001:38:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:38:mod xmmreg1 r/m
VPMINSD — Minimum of Packed Signed Dword Integers	
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_2: w xmmreg2 001:39:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:39:mod xmmreg1 r/m
VPMINUD — Minimum of Packed Unsigned Dword Integers	
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_2: w xmmreg2 001:3B:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:3B:mod xmmreg1 r/m
VPMINUW — Minimum of Packed Unsigned Word Integers	
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_2: w xmmreg2 001:3A:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:3A:mod xmmreg1 r/m
VPMOVSXBD — Packed Move Sign Extend - Byte to Dword	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:21:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:21:mod xmmreg1 r/m
VPMOVSXBQ — Packed Move Sign Extend - Byte to Qword	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:22:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:22:mod xmmreg1 r/m
VPMOVSXBW — Packed Move Sign Extend - Byte to Word	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:20:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:20:mod xmmreg1 r/m
VPMOVSXWD — Packed Move Sign Extend - Word to Dword	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:23:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:23:mod xmmreg1 r/m
VPMOVSXWQ — Packed Move Sign Extend - Word to Qword	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:24:11 xmmreg1 xmmreg2

Instruction and Format	Encoding
mem to xmmreg1	C4: rxb0_2: w_F 001:24:mod xmmreg1 r/m
VPMOVSXDQ — Packed Move Sign Extend - Dword to Qword	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:25:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:25:mod xmmreg1 r/m
VPMOVZXBD — Packed Move Zero Extend - Byte to Dword	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:31:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:31:mod xmmreg1 r/m
VPMOVZXBQ — Packed Move Zero Extend - Byte to Qword	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:32:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:32:mod xmmreg1 r/m
VPMOVZXBW — Packed Move Zero Extend - Byte to Word	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:30:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:30:mod xmmreg1 r/m
VPMOVZXWD — Packed Move Zero Extend - Word to Dword	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:33:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:33:mod xmmreg1 r/m
VPMOVZXWQ — Packed Move Zero Extend - Word to Qword	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:34:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:34:mod xmmreg1 r/m
VPMOVZXDQ — Packed Move Zero Extend - Dword to Qword	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:35:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:35:mod xmmreg1 r/m
VPMULDQ — Multiply Packed Signed Dword Integers	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:28:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:28:mod xmmreg1 r/m
VPMULLD — Multiply Packed Signed Dword Integers, Store low Result	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:40:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:40:mod xmmreg1 r/m
VPTEST — Logical Compare	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:17:11 xmmreg1 xmmreg2
mem to xmmreg	C4: rxb0_2: w_F 001:17:mod xmmreg1 r/m
ymmreg2 to ymmreg1	C4: rxb0_2: w_F 101:17:11 ymmreg1 ymmreg2
mem to ymmreg	C4: rxb0_2: w_F 101:17:mod ymmreg1 r/m
VROUNDPD — Round Packed Double Precision Values	
xmmreg2 to xmmreg1, imm8	C4: rxb0_3: w_F 001:09:11 xmmreg1 xmmreg2: imm
mem to xmmreg1, imm8	C4: rxb0_3: w_F 001:09:mod xmmreg1 r/m: imm
ymmreg2 to ymmreg1, imm8	C4: rxb0_3: w_F 101:09:11 ymmreg1 ymmreg2: imm
mem to ymmreg1, imm8	C4: rxb0_3: w_F 101:09:mod ymmreg1 r/m: imm
VROUNDPS — Round Packed Single Precision Values	

Instruction and Format	Encoding
xmmreg2 to xmmreg1, imm8	C4: rxb0_3: w_F 001:08:11 xmmreg1 xmmreg2: imm
mem to xmmreg1, imm8	C4: rxb0_3: w_F 001:08:mod xmmreg1 r/m: imm
ymmreg2 to ymmreg1, imm8	C4: rxb0_3: w_F 101:08:11 ymmreg1 ymmreg2: imm
mem to ymmreg1, imm8	C4: rxb0_3: w_F 101:08:mod ymmreg1 r/m: imm
VROUNDSD — Round Scalar Double Precision Value	
xmmreg2 and xmmreg3 to xmmreg1, imm8	C4: rxb0_3: w xmmreg2 001:0B:11 xmmreg1 xmmreg3: imm
xmmreg2 and mem to xmmreg1, imm8	C4: rxb0_3: w xmmreg2 001:0B:mod xmmreg1 r/m: imm
VROUNDSS — Round Scalar Single Precision Value	
xmmreg2 and xmmreg3 to xmmreg1, imm8	C4: rxb0_3: w xmmreg2 001:0A:11 xmmreg1 xmmreg3: imm
xmmreg2 and mem to xmmreg1, imm8	C4: rxb0_3: w xmmreg2 001:0A:mod xmmreg1 r/m: imm
VPCMPESTRI — Packed Compare Explicit Length Strings, Return Index	
xmmreg2 with xmmreg1, imm8	C4: rxb0_3: w_F 001:61:11 xmmreg1 xmmreg2: imm
mem with xmmreg1, imm8	C4: rxb0_3: w_F 001:61:mod xmmreg1 r/m: imm
VPCMPESTRM — Packed Compare Explicit Length Strings, Return Mask	
xmmreg2 with xmmreg1, imm8	C4: rxb0_3: w_F 001:60:11 xmmreg1 xmmreg2: imm
mem with xmmreg1, imm8	C4: rxb0_3: w_F 001:60:mod xmmreg1 r/m: imm
VPCMPGTQ — Compare Packed Data for Greater Than	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:28:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:28:mod xmmreg1 r/m
VPCMPISTRI — Packed Compare Implicit Length Strings, Return Index	
xmmreg2 with xmmreg1, imm8	C4: rxb0_3: w_F 001:63:11 xmmreg1 xmmreg2: imm
mem with xmmreg1, imm8	C4: rxb0_3: w_F 001:63:mod xmmreg1 r/m: imm
VPCMPISTRM — Packed Compare Implicit Length Strings, Return Mask	
xmmreg2 with xmmreg1, imm8	C4: rxb0_3: w_F 001:62:11 xmmreg1 xmmreg2: imm
mem with xmmreg, imm8	C4: rxb0_3: w_F 001:62:mod xmmreg1 r/m: imm
VAESDEC — Perform One Round of an AES Decryption Flow	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:DE:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:DE:mod xmmreg1 r/m
VAESDECLAST — Perform Last Round of an AES Decryption Flow	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:DF:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:DF:mod xmmreg1 r/m
VAESENC — Perform One Round of an AES Encryption Flow	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:DC:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:DC:mod xmmreg1 r/m
VAESENCLAST — Perform Last Round of an AES Encryption Flow	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:DD:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:DD:mod xmmreg1 r/m
VAESIMC — Perform the AES InvMixColumn Transformation	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:DB:11 xmmreg1 xmmreg2

Instruction and Format	Encoding
mem to xmmreg1	C4: rxb0_2: w_F 001:DB:mod xmmreg1 r/m
VAESKEYGENASSIST — AES Round Key Generation Assist	
xmmreg2 to xmmreg1, imm8	C4: rxb0_3: w_F 001:DF:11 xmmreg1 xmmreg2: imm
mem to xmmreg, imm8	C4: rxb0_3: w_F 001:DF:mod xmmreg1 r/m: imm
VPABSB — Packed Absolute Value	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:1C:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:1C:mod xmmreg1 r/m
VPABSD — Packed Absolute Value	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:1E:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:1E:mod xmmreg1 r/m
VPABSW — Packed Absolute Value	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:1D:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:1D:mod xmmreg1 r/m
VPALIGNR — Packed Align Right	
xmmreg2 with xmmreg3 to xmmreg1, imm8	C4: rxb0_3: w xmmreg2 001:DD:11 xmmreg1 xmmreg3: imm
xmmreg2 with mem to xmmreg1, imm8	C4: rxb0_3: w xmmreg2 001:DD:mod xmmreg1 r/m: imm
VPHADDD — Packed Horizontal Add	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:02:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:02:mod xmmreg1 r/m
VPHADDW — Packed Horizontal Add	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:01:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:01:mod xmmreg1 r/m
VPHADDSW — Packed Horizontal Add and Saturate	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:03:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:03:mod xmmreg1 r/m
VPHSUBD — Packed Horizontal Subtract	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:06:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:06:mod xmmreg1 r/m
VPHSUBW — Packed Horizontal Subtract	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:05:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:05:mod xmmreg1 r/m
VPHSUBSW — Packed Horizontal Subtract and Saturate	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:07:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:07:mod xmmreg1 r/m
VPMADDUBSW — Multiply and Add Packed Signed and Unsigned Bytes	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:04:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:04:mod xmmreg1 r/m
VPMULHRSW — Packed Multiply High with Round and Scale	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:0B:11 xmmreg1 xmmreg3

Instruction and Format	Encoding
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:0B:mod xmmreg1 r/m
VPSHUFB — Packed Shuffle Bytes	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:00:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:00:mod xmmreg1 r/m
VPSIGNB — Packed SIGN	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:08:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:08:mod xmmreg1 r/m
VPSIGND — Packed SIGN	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:0A:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:0A:mod xmmreg1 r/m
VPSIGNW — Packed SIGN	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:09:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:09:mod xmmreg1 r/m
VADDSUBPD — Packed Double-FP Add/Subtract	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:D0:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:D0:mod xmmreg1 r/m
xmmreglo2[1] with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:D0:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:D0:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:D0:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:D0:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:D0:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:D0:mod ymmreg1 r/m
VADDSUBPS — Packed Single-FP Add/Subtract	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:D0:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 011:D0:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:D0:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 011:D0:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 111:D0:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 111:D0:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 111:D0:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 111:D0:mod ymmreg1 r/m
VHADDPD — Packed Double-FP Horizontal Add	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:7C:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:7C:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:7C:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:7C:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:7C:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:7C:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:7C:11 ymmreg1 ymmreglo3

Instruction and Format	Encoding
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:7C:mod ymmreg1 r/m
VHADDPS — Packed Single-FP Horizontal Add	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:7C:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 011:7C:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:7C:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 011:7C:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 111:7C:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 111:7C:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 111:7C:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 111:7C:mod ymmreg1 r/m
VHSUBPD — Packed Double-FP Horizontal Subtract	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:7D:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:7D:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:7D:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:7D:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:7D:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:7D:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:7D:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:7D:mod ymmreg1 r/m
VHSUBPS — Packed Single-FP Horizontal Subtract	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:7D:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 011:7D:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:7D:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 011:7D:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 111:7D:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 111:7D:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 111:7D:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 111:7D:mod ymmreg1 r/m
VLDDQU — Load Unaligned Integer 128 Bits	
mem to xmmreg1	C4: rxb0_1: w_F 011:F0:mod xmmreg1 r/m
mem to xmmreg1	C5: r_F 011:F0:mod xmmreg1 r/m
mem to ymmreg1	C4: rxb0_1: w_F 111:F0:mod ymmreg1 r/m
mem to ymmreg1	C5: r_F 111:F0:mod ymmreg1 r/m
VMOVDDUP — Move One Double-FP and Duplicate	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 011:12:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 011:12:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 011:12:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 011:12:mod xmmreg1 r/m
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 111:12:11 ymmreg1 ymmreg2

Instruction and Format	Encoding
mem to ymmreg1	C4: rxb0_1: w_F 111:12:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_ F 111:12:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 111:12:mod ymmreg1 r/m
VMOVHLPS — Move Packed Single Precision Floating-Point Values High to Low	
xmmreg2 and xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:12:11 xmmreg1 xmmreg3
xmmreglo2 and xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:12:11 xmmreg1 xmmreglo3
VMOVSHDUP — Move Packed Single-FP High and Duplicate	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 010:16:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 010:16:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 010:16:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 010:16:mod xmmreg1 r/m
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 110:16:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 110:16:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 110:16:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 110:16:mod ymmreg1 r/m
VMOVSLDUP — Move Packed Single-FP Low and Duplicate	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 010:12:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 010:12:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 010:12:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 010:12:mod xmmreg1 r/m
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 110:12:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 110:12:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 110:12:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 110:12:mod ymmreg1 r/m
VADDPD — Add Packed Double Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:58:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:58:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:58:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:58:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:58:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:58:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:58:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:58:mod ymmreg1 r/m
VADDSD — Add Scalar Double Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:58:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 011:58:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:58:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5 r_xmmreglo2 011:58:mod xmmreg1 r/m
VANDPD — Bitwise Logical AND of Packed Double Precision Floating-Point Values	

Instruction and Format	Encoding
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:54:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:54:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:54:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:54:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:54:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:54:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:54:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:54:mod ymmreg1 r/m
VANDNPD — Bitwise Logical AND NOT of Packed Double Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:55:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:55:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:55:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:55:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:55:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:55:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:55:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:55:mod ymmreg1 r/m
VCMPPD — Compare Packed Double Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:C2:11 xmmreg1 xmmreg3: imm
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:C2:mod xmmreg1 r/m: imm
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:C2:11 xmmreg1 xmmreglo3: imm
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:C2:mod xmmreg1 r/m: imm
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:C2:11 ymmreg1 ymmreg3: imm
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:C2:mod ymmreg1 r/m: imm
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:C2:11 ymmreg1 ymmreglo3: imm
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:C2:mod ymmreg1 r/m: imm
VCMPSD — Compare Scalar Double Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:C2:11 xmmreg1 xmmreg3: imm
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 011:C2:mod xmmreg1 r/m: imm
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:C2:11 xmmreg1 xmmreglo3: imm
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 011:C2:mod xmmreg1 r/m: imm
VCOMISD — Compare Scalar Ordered Double Precision Floating-Point Values and Set EFLAGS	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 001:2F:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 001:2F:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 001:2F:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 001:2F:mod xmmreg1 r/m
VCVTDQ2PD— Convert Packed Dword Integers to Packed Double Precision FP Values	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 010:E6:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 010:E6:mod xmmreg1 r/m

Instruction and Format	Encoding
xmmreglo to xmmreg1	C5: r_F 010:E6:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 010:E6:mod xmmreg1 r/m
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 110:E6:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 110:E6:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 110:E6:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 110:E6:mod ymmreg1 r/m
VCVTDQ2PS— Convert Packed Dword Integers to Packed Single Precision FP Values	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 000:5B:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 000:5B:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 000:5B:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 000:5B:mod xmmreg1 r/m
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 100:5B:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 100:5B:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 100:5B:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 100:5B:mod ymmreg1 r/m
VCVTPD2DQ— Convert Packed Double Precision FP Values to Packed Dword Integers	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 011:E6:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 011:E6:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 011:E6:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 011:E6:mod xmmreg1 r/m
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 111:E6:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 111:E6:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 111:E6:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 111:E6:mod ymmreg1 r/m
VCVTPD2PS— Convert Packed Double Precision FP Values to Packed Single Precision FP Values	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 001:5A:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 001:5A:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 001:5A:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 001:5A:mod xmmreg1 r/m
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 101:5A:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 101:5A:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 101:5A:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 101:5A:mod ymmreg1 r/m
VCVTPS2DQ— Convert Packed Single Precision FP Values to Packed Dword Integers	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 001:5B:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 001:5B:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 001:5B:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 001:5B:mod xmmreg1 r/m
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 101:5B:11 ymmreg1 ymmreg2

Instruction and Format	Encoding
mem to ymmreg1	C4: rxb0_1: w_F 101:5B:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 101:5B:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 101:5B:mod ymmreg1 r/m
VCVTPS2PD— Convert Packed Single Precision FP Values to Packed Double Precision FP Values	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 000:5A:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 000:5A:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 000:5A:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 000:5A:mod xmmreg1 r/m
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 100:5A:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 100:5A:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 100:5A:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 100:5A:mod ymmreg1 r/m
VCVTSD2SI— Convert Scalar Double Precision FP Value to Signed Integer	
xmmreg1 to reg32	C4: rxb0_1: 0_F 011:2D:11 reg xmmreg1
mem to reg32	C4: rxb0_1: 0_F 011:2D:mod reg r/m
xmmreglo to reg32	C5: r_F 011:2D:11 reg xmmreglo
mem to reg32	C5: r_F 011:2D:mod reg r/m
ymmreg1 to reg64	C4: rxb0_1: 1_F 111:2D:11 reg ymmreg1
mem to reg64	C4: rxb0_1: 1_F 111:2D:mod reg r/m
VCVTSD2SS — Convert Scalar Double Precision FP Value to Scalar Single Precision FP Value	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:5A:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 011:5A:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:5A:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 011:5A:mod xmmreg1 r/m
VCVTSI2SD— Convert Signed Integer to Scalar Double Precision FP Value	
xmmreg2 with reg to xmmreg1	C4: rxb0_1: 0 xmmreg2 011:2A:11 xmmreg1 reg
xmmreg2 with mem to xmmreg1	C4: rxb0_1: 0 xmmreg2 011:2A:mod xmmreg1 r/m
xmmreglo2 with reglo to xmmreg1	C5: r_xmmreglo2 011:2A:11 xmmreg1 reglo
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 011:2A:mod xmmreg1 r/m
ymmreg2 with reg to ymmreg1	C4: rxb0_1: 1 ymmreg2 111:2A:11 ymmreg1 reg
ymmreg2 with mem to ymmreg1	C4: rxb0_1: 1 ymmreg2 111:2A:mod ymmreg1 r/m
VCVTSS2SD — Convert Scalar Single Precision FP Value to Scalar Double Precision FP Value	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:5A:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 010:5A:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 010:5A:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 010:5A:mod xmmreg1 r/m
VCVTTPD2DQ— Convert with Truncation Packed Double Precision FP Values to Packed Dword Integers	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 001:E6:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 001:E6:mod xmmreg1 r/m

Instruction and Format	Encoding
xmmreglo to xmmreg1	C5: r_F 001:E6:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 001:E6:mod xmmreg1 r/m
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 101:E6:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 101:E6:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 101:E6:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 101:E6:mod ymmreg1 r/m
VCVTTPS2DQ— Convert with Truncation Packed Single Precision FP Values to Packed Dword Integers	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 010:5B:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 010:5B:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 010:5B:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 010:5B:mod xmmreg1 r/m
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 110:5B:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 110:5B:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 110:5B:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 110:5B:mod ymmreg1 r/m
VCVTTSD2SI— Convert with Truncation Scalar Double Precision FP Value to Signed Integer	
xmmreg1 to reg32	C4: rxb0_1: 0_F 011:2C:11 reg xmmreg1
mem to reg32	C4: rxb0_1: 0_F 011:2C:mod reg r/m
xmmreglo to reg32	C5: r_F 011:2C:11 reg xmmreglo
mem to reg32	C5: r_F 011:2C:mod reg r/m
xmmreg1 to reg64	C4: rxb0_1: 1_F 011:2C:11 reg xmmreg1
mem to reg64	C4: rxb0_1: 1_F 011:2C:mod reg r/m
VDIVPD — Divide Packed Double Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:5E:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:5E:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:5E:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:5E:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:5E:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:5E:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:5E:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:5E:mod ymmreg1 r/m
VDIVSD — Divide Scalar Double Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:5E:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 011:5E:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:5E:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 011:5E:mod xmmreg1 r/m
VMASKMOVDQU— Store Selected Bytes of Double Quadword	
xmmreg1 to mem; xmmreg2 as mask	C4: rxb0_1: w_F 001:F7:11 r/m xmmreg1: xmmreg2
xmmreg1 to mem; xmmreg2 as mask	C5: r_F 001:F7:11 r/m xmmreg1: xmmreg2

Instruction and Format	Encoding
VMAXPD — Return Maximum Packed Double Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:5F:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:5F:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:5F:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:5F:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:5F:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:5F:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:5F:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:5F:mod ymmreg1 r/m
VMAXSD — Return Maximum Scalar Double Precision Floating-Point Value	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:5F:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 011:5F:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:5F:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 011:5F:mod xmmreg1 r/m
VMINPD — Return Minimum Packed Double Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:5D:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:5D:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:5D:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:5D:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:5D:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:5D:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:5D:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:5D:mod ymmreg1 r/m
VMINSD — Return Minimum Scalar Double Precision Floating-Point Value	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:5D:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 011:5D:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:5D:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 011:5D:mod xmmreg1 r/m
VMOVAPD — Move Aligned Packed Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 001:28:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 001:28:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 001:28:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 001:28:mod xmmreg1 r/m
xmmreg1 to xmmreg2	C4: rxb0_1: w_F 001:29:11 xmmreg2 xmmreg1
xmmreg1 to mem	C4: rxb0_1: w_F 001:29:mod r/m xmmreg1
xmmreg1 to xmmreglo	C5: r_F 001:29:11 xmmreglo xmmreg1
xmmreg1 to mem	C5: r_F 001:29:mod r/m xmmreg1
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 101:28:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 101:28:mod ymmreg1 r/m

Instruction and Format	Encoding
ymmreglo to ymmreg1	C5: r_F 101:28:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 101:28:mod ymmreg1 r/m
ymmreg1 to ymmreg2	C4: rxb0_1: w_F 101:29:11 ymmreg2 ymmreg1
ymmreg1 to mem	C4: rxb0_1: w_F 101:29:mod r/m ymmreg1
ymmreg1 to ymmreglo	C5: r_F 101:29:11 ymmreglo ymmreg1
ymmreg1 to mem	C5: r_F 101:29:mod r/m ymmreg1
VMOVD — Move Doubleword	
reg32 to xmmreg1	C4: rxb0_1: 0_F 001:6E:11 xmmreg1 reg32
mem32 to xmmreg1	C4: rxb0_1: 0_F 001:6E:mod xmmreg1 r/m
reg32 to xmmreg1	C5: r_F 001:6E:11 xmmreg1 reg32
mem32 to xmmreg1	C5: r_F 001:6E:mod xmmreg1 r/m
xmmreg1 to reg32	C4: rxb0_1: 0_F 001:7E:11 reg32 xmmreg1
xmmreg1 to mem32	C4: rxb0_1: 0_F 001:7E:mod mem32 xmmreg1
xmmreglo to reg32	C5: r_F 001:7E:11 reg32 xmmreglo
xmmreglo to mem32	C5: r_F 001:7E:mod mem32 xmmreglo
VMOVQ — Move Quadword	
reg64 to xmmreg1	C4: rxb0_1: 1_F 001:6E:11 xmmreg1 reg64
mem64 to xmmreg1	C4: rxb0_1: 1_F 001:6E:mod xmmreg1 r/m
xmmreg1 to reg64	C4: rxb0_1: 1_F 001:7E:11 reg64 xmmreg1
xmmreg1 to mem64	C4: rxb0_1: 1_F 001:7E:mod r/m xmmreg1
VMOVDQA — Move Aligned Double Quadword	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 001:6F:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 001:6F:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 001:6F:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 001:6F:mod xmmreg1 r/m
xmmreg1 to xmmreg2	C4: rxb0_1: w_F 001:7F:11 xmmreg2 xmmreg1
xmmreg1 to mem	C4: rxb0_1: w_F 001:7F:mod r/m xmmreg1
xmmreg1 to xmmreglo	C5: r_F 001:7F:11 xmmreglo xmmreg1
xmmreg1 to mem	C5: r_F 001:7F:mod r/m xmmreg1
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 101:6F:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 101:6F:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 101:6F:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 101:6F:mod ymmreg1 r/m
ymmreg1 to ymmreg2	C4: rxb0_1: w_F 101:7F:11 ymmreg2 ymmreg1
ymmreg1 to mem	C4: rxb0_1: w_F 101:7F:mod r/m ymmreg1
ymmreg1 to ymmreglo	C5: r_F 101:7F:11 ymmreglo ymmreg1
ymmreg1 to mem	C5: r_F 101:7F:mod r/m ymmreg1
VMOVDQU — Move Unaligned Double Quadword	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 010:6F:11 xmmreg1 xmmreg2

Instruction and Format	Encoding
mem to xmmreg1	C4: rxb0_1: w_F 010:6F:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 010:6F:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 010:6F:mod xmmreg1 r/m
xmmreg1 to xmmreg2	C4: rxb0_1: w_F 010:7F:11 xmmreg2 xmmreg1
xmmreg1 to mem	C4: rxb0_1: w_F 010:7F:mod r/m xmmreg1
xmmreg1 to xmmreglo	C5: r_F 010:7F:11 xmmreglo xmmreg1
xmmreg1 to mem	C5: r_F 010:7F:mod r/m xmmreg1
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 110:6F:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 110:6F:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 110:6F:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 110:6F:mod ymmreg1 r/m
ymmreg1 to ymmreg2	C4: rxb0_1: w_F 110:7F:11 ymmreg2 ymmreg1
ymmreg1 to mem	C4: rxb0_1: w_F 110:7F:mod r/m ymmreg1
ymmreg1 to ymmreglo	C5: r_F 110:7F:11 ymmreglo ymmreg1
ymmreg1 to mem	C5: r_F 110:7F:mod r/m ymmreg1
VMOVHPD — Move High Packed Double Precision Floating-Point Value	
xmmreg1 and mem to xmmreg2	C4: rxb0_1: w xmmreg1 001:16:11 xmmreg2 r/m
xmmreg1 and mem to xmmreglo2	C5: r_xmmreg1 001:16:11 xmmreglo2 r/m
xmmreg1 to mem	C4: rxb0_1: w_F 001:17:mod r/m xmmreg1
xmmreglo to mem	C5: r_F 001:17:mod r/m xmmreglo
VMOVLPD — Move Low Packed Double Precision Floating-Point Value	
xmmreg1 and mem to xmmreg2	C4: rxb0_1: w xmmreg1 001:12:11 xmmreg2 r/m
xmmreg1 and mem to xmmreglo2	C5: r_xmmreg1 001:12:11 xmmreglo2 r/m
xmmreg1 to mem	C4: rxb0_1: w_F 001:13:mod r/m xmmreg1
xmmreglo to mem	C5: r_F 001:13:mod r/m xmmreglo
VMOVMSKPD — Extract Packed Double Precision Floating-Point Sign Mask	
xmmreg2 to reg	C4: rxb0_1: w_F 001:50:11 reg xmmreg1
xmmreglo to reg	C5: r_F 001:50:11 reg xmmreglo
ymmreg2 to reg	C4: rxb0_1: w_F 101:50:11 reg ymmreg1
ymmreglo to reg	C5: r_F 101:50:11 reg ymmreglo
VMOVNTDQ — Store Double Quadword Using Non-Temporal Hint	
xmmreg1 to mem	C4: rxb0_1: w_F 001:E7:11 r/m xmmreg1
xmmreglo to mem	C5: r_F 001:E7:11 r/m xmmreglo
ymmreg1 to mem	C4: rxb0_1: w_F 101:E7:11 r/m ymmreg1
ymmreglo to mem	C5: r_F 101:E7:11 r/m ymmreglo
VMOVNTPD — Store Packed Double Precision Floating-Point Values Using Non-Temporal Hint	
xmmreg1 to mem	C4: rxb0_1: w_F 001:2B:11 r/m xmmreg1
xmmreglo to mem	C5: r_F 001:2B:11 r/m xmmreglo
ymmreg1 to mem	C4: rxb0_1: w_F 101:2B:11r/m ymmreg1

Instruction and Format	Encoding
ymmreglo to mem	C5: r_F 101:2B:11r/m ymmreglo
VMOVSD — Move Scalar Double Precision Floating-Point Value	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:10:11 xmmreg1 xmmreg3
mem to xmmreg1	C4: rxb0_1: w_F 011:10:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:10:11 xmmreg1 xmmreglo3
mem to xmmreg1	C5: r_F 011:10:mod xmmreg1 r/m
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:11:11 xmmreg1 xmmreg3
xmmreg1 to mem	C4: rxb0_1: w_F 011:11:mod r/m xmmreg1
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:11:11 xmmreg1 xmmreglo3
xmmreglo to mem	C5: r_F 011:11:mod r/m xmmreglo
VMOVUPD — Move Unaligned Packed Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 001:10:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 001:10:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 001:10:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 001:10:mod xmmreg1 r/m
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 101:10:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 101:10:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 101:10:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 101:10:mod ymmreg1 r/m
xmmreg1 to xmmreg2	C4: rxb0_1: w_F 001:11:11 xmmreg2 xmmreg1
xmmreg1 to mem	C4: rxb0_1: w_F 001:11:mod r/m xmmreg1
xmmreg1 to xmmreglo	C5: r_F 001:11:11 xmmreglo xmmreg1
xmmreg1 to mem	C5: r_F 001:11:mod r/m xmmreg1
ymmreg1 to ymmreg2	C4: rxb0_1: w_F 101:11:11 ymmreg2 ymmreg1
ymmreg1 to mem	C4: rxb0_1: w_F 101:11:mod r/m ymmreg1
ymmreg1 to ymmreglo	C5: r_F 101:11:11 ymmreglo ymmreg1
ymmreg1 to mem	C5: r_F 101:11:mod r/m ymmreg1
VMULPD — Multiply Packed Double Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:59:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:59:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:59:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:59:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:59:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:59:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:59:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:59:mod ymmreg1 r/m
VMULSD — Multiply Scalar Double Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:59:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 011:59:mod xmmreg1 r/m

Instruction and Format	Encoding
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:59:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 011:59:mod xmmreg1 r/m
VORPD — Bitwise Logical OR of Double Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:56:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:56:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:56:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:56:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:56:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:56:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:56:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:56:mod ymmreg1 r/m
VPACKSSWB— Pack with Signed Saturation	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:63:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:63:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:63:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:63:mod xmmreg1 r/m
VPACKSSDW— Pack with Signed Saturation	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:6B:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:6B:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:6B:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:6B:mod xmmreg1 r/m
VPACKUSWB— Pack with Unsigned Saturation	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:67:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:67:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:67:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:67:mod xmmreg1 r/m
VPADDB — Add Packed Integers	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:FC:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:FC:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:FC:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:FC:mod xmmreg1 r/m
VPADDW — Add Packed Integers	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:FD:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:FD:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:FD:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:FD:mod xmmreg1 r/m
VPADDD — Add Packed Integers	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:FE:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:FE:mod xmmreg1 r/m

Instruction and Format	Encoding
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:FE:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:FE:mod xmmreg1 r/m
VPADDQ — Add Packed Quadword Integers	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:D4:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:D4:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:D4:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:D4:mod xmmreg1 r/m
VPADDSB — Add Packed Signed Integers with Signed Saturation	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:EC:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:EC:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:EC:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:EC:mod xmmreg1 r/m
VPADDSW — Add Packed Signed Integers with Signed Saturation	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:ED:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:ED:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:ED:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:ED:mod xmmreg1 r/m
VPADDUSB — Add Packed Unsigned Integers with Unsigned Saturation	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:DC:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:DC:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:DC:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:DC:mod xmmreg1 r/m
VPADDUSW — Add Packed Unsigned Integers with Unsigned Saturation	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:DD:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:DD:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:DD:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:DD:mod xmmreg1 r/m
VPAND — Logical AND	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:DB:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:DB:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:DB:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:DB:mod xmmreg1 r/m
VPANDN — Logical AND NOT	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:DF:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:DF:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:DF:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:DF:mod xmmreg1 r/m
VPAVGB — Average Packed Integers	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:E0:11 xmmreg1 xmmreg3

Instruction and Format	Encoding
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:E0:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:E0:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:E0:mod xmmreg1 r/m
VPAVGW — Average Packed Integers	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:E3:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:E3:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:E3:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:E3:mod xmmreg1 r/m
VPCMPEQB — Compare Packed Data for Equal	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:74:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:74:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:74:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:74:mod xmmreg1 r/m
VPCMPEQW — Compare Packed Data for Equal	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:75:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:75:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:75:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:75:mod xmmreg1 r/m
VPCMPEQD — Compare Packed Data for Equal	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:76:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:76:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:76:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:76:mod xmmreg1 r/m
VPCMPGTB — Compare Packed Signed Integers for Greater Than	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:64:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:64:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:64:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:64:mod xmmreg1 r/m
VPCMPGTW — Compare Packed Signed Integers for Greater Than	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:65:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:65:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:65:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:65:mod xmmreg1 r/m
VPCMPGTD — Compare Packed Signed Integers for Greater Than	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:66:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:66:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:66:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:66:mod xmmreg1 r/m
VPEXTRW — Extract Word	

Instruction and Format	Encoding
xmmreg1 to reg using imm	C4: rxb0_1: 0_F 001:C5:11 reg xmmreg1: imm
xmmreg1 to reg using imm	C5: r_F 001:C5:11 reg xmmreg1: imm
VPINSRW — Insert Word	
xmmreg2 with reg to xmmreg1	C4: rxb0_1: 0 xmmreg2 001:C4:11 xmmreg1 reg: imm
xmmreg2 with mem to xmmreg1	C4: rxb0_1: 0 xmmreg2 001:C4:mod xmmreg1 r/m: imm
xmmreglo2 with reglo to xmmreg1	C5: r_xmmreglo2 001:C4:11 xmmreg1 reglo: imm
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:C4:mod xmmreg1 r/m: imm
VPMADDWD — Multiply and Add Packed Integers	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:F5:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:F5:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:F5:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:F5:mod xmmreg1 r/m
VPMAXSW — Maximum of Packed Signed Word Integers	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:EE:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:EE:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:EE:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:EE:mod xmmreg1 r/m
VPMAXUB — Maximum of Packed Unsigned Byte Integers	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:DE:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:DE:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:DE:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:DE:mod xmmreg1 r/m
VPMINSW — Minimum of Packed Signed Word Integers	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:EA:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:EA:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:EA:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:EA:mod xmmreg1 r/m
VPMINUB — Minimum of Packed Unsigned Byte Integers	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:DA:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:DA:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:DA:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:DA:mod xmmreg1 r/m
VPMOVMSKB — Move Byte Mask	
xmmreg1 to reg	C4: rxb0_1: w_F 001:D7:11 reg xmmreg1
xmmreg1 to reg	C5: r_F 001:D7:11 reg xmmreg1
VPMULHUW — Multiply Packed Unsigned Integers and Store High Result	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:E4:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:E4:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:E4:11 xmmreg1 xmmreglo3

Instruction and Format	Encoding
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:E4:mod xmmreg1 r/m
VPMULHW — Multiply Packed Signed Integers and Store High Result	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:E5:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:E5:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:E5:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:E5:mod xmmreg1 r/m
VPMULLW — Multiply Packed Signed Integers and Store Low Result	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:D5:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:D5:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:D5:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:D5:mod xmmreg1 r/m
VPMULUDQ — Multiply Packed Unsigned Doubleword Integers	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:F4:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:F4:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:F4:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:F4:mod xmmreg1 r/m
VPOR — Bitwise Logical OR	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:EB:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:EB:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:EB:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:EB:mod xmmreg1 r/m
VPSADBW — Compute Sum of Absolute Differences	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:F6:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:F6:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:F6:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:F6:mod xmmreg1 r/m
VPSHUFD — Shuffle Packed Doublewords	
xmmreg2 to xmmreg1 using imm	C4: rxb0_1: w_F 001:70:11 xmmreg1 xmmreg2: imm
mem to xmmreg1 using imm	C4: rxb0_1: w_F 001:70:mod xmmreg1 r/m: imm
xmmreglo to xmmreg1 using imm	C5: r_F 001:70:11 xmmreg1 xmmreglo: imm
mem to xmmreg1 using imm	C5: r_F 001:70:mod xmmreg1 r/m: imm
VPSHUFHW — Shuffle Packed High Words	
xmmreg2 to xmmreg1 using imm	C4: rxb0_1: w_F 010:70:11 xmmreg1 xmmreg2: imm
mem to xmmreg1 using imm	C4: rxb0_1: w_F 010:70:mod xmmreg1 r/m: imm
xmmreglo to xmmreg1 using imm	C5: r_F 010:70:11 xmmreg1 xmmreglo: imm
mem to xmmreg1 using imm	C5: r_F 010:70:mod xmmreg1 r/m: imm
VPSHUFLW — Shuffle Packed Low Words	
xmmreg2 to xmmreg1 using imm	C4: rxb0_1: w_F 011:70:11 xmmreg1 xmmreg2: imm
mem to xmmreg1 using imm	C4: rxb0_1: w_F 011:70:mod xmmreg1 r/m: imm

Instruction and Format	Encoding
xmmreglo to xmmreg1 using imm	C5: r_F 011:70:11 xmmreg1 xmmreglo: imm
mem to xmmreg1 using imm	C5: r_F 011:70:mod xmmreg1 r/m: imm
VPSLLDQ — Shift Double Quadword Left Logical	
xmmreg2 to xmmreg1 using imm	C4: rxb0_1: w_F 001:73:11 xmmreg1 xmmreg2: imm
xmmreglo to xmmreg1 using imm	C5: r_F 001:73:11 xmmreg1 xmmreglo: imm
VPSLLW — Shift Packed Data Left Logical	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:F1:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:F1:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:F1:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:F1:mod xmmreg1 r/m
xmmreg2 to xmmreg1 using imm8	C4: rxb0_1: w_F 001:71:11 xmmreg1 xmmreg2: imm
xmmreglo to xmmreg1 using imm8	C5: r_F 001:71:11 xmmreg1 xmmreglo: imm
VPSLLD — Shift Packed Data Left Logical	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:F2:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:F2:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:F2:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:F2:mod xmmreg1 r/m
xmmreg2 to xmmreg1 using imm8	C4: rxb0_1: w_F 001:72:11 xmmreg1 xmmreg2: imm
xmmreglo to xmmreg1 using imm8	C5: r_F 001:72:11 xmmreg1 xmmreglo: imm
VPSLLQ — Shift Packed Data Left Logical	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:F3:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:F3:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:F3:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:F3:mod xmmreg1 r/m
xmmreg2 to xmmreg1 using imm8	C4: rxb0_1: w_F 001:73:11 xmmreg1 xmmreg2: imm
xmmreglo to xmmreg1 using imm8	C5: r_F 001:73:11 xmmreg1 xmmreglo: imm
VPSRAW — Shift Packed Data Right Arithmetic	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:E1:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:E1:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:E1:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:E1:mod xmmreg1 r/m
xmmreg2 to xmmreg1 using imm8	C4: rxb0_1: w_F 001:71:11 xmmreg1 xmmreg2: imm
xmmreglo to xmmreg1 using imm8	C5: r_F 001:71:11 xmmreg1 xmmreglo: imm
VPSRAD — Shift Packed Data Right Arithmetic	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:E2:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:E2:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:E2:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:E2:mod xmmreg1 r/m
xmmreg2 to xmmreg1 using imm8	C4: rxb0_1: w_F 001:72:11 xmmreg1 xmmreg2: imm

Instruction and Format	Encoding
xmmreglo to xmmreg1 using imm8	C5: r_F 001:72:11 xmmreg1 xmmreglo: imm
VPSRLDQ — Shift Double Quadword Right Logical	
xmmreg2 to xmmreg1 using imm8	C4: rxb0_1: w_F 001:73:11 xmmreg1 xmmreg2: imm
xmmreglo to xmmreg1 using imm8	C5: r_F 001:73:11 xmmreg1 xmmreglo: imm
VPSRLW — Shift Packed Data Right Logical	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:D1:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:D1:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:D1:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:D1:mod xmmreg1 r/m
xmmreg2 to xmmreg1 using imm8	C4: rxb0_1: w_F 001:71:11 xmmreg1 xmmreg2: imm
xmmreglo to xmmreg1 using imm8	C5: r_F 001:71:11 xmmreg1 xmmreglo: imm
VPSRLD — Shift Packed Data Right Logical	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:D2:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:D2:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:D2:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:D2:mod xmmreg1 r/m
xmmreg2 to xmmreg1 using imm8	C4: rxb0_1: w_F 001:72:11 xmmreg1 xmmreg2: imm
xmmreglo to xmmreg1 using imm8	C5: r_F 001:72:11 xmmreg1 xmmreglo: imm
VPSRLQ — Shift Packed Data Right Logical	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:D3:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:D3:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:D3:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:D3:mod xmmreg1 r/m
xmmreg2 to xmmreg1 using imm8	C4: rxb0_1: w_F 001:73:11 xmmreg1 xmmreg2: imm
xmmreglo to xmmreg1 using imm8	C5: r_F 001:73:11 xmmreg1 xmmreglo: imm
VPSUBB — Subtract Packed Integers	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:F8:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:F8:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:F8:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:F8:mod xmmreg1 r/m
VPSUBW — Subtract Packed Integers	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:F9:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:F9:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:F9:11 xmmreg1 xmmreglo3
xmmrelog2 with mem to xmmreg1	C5: r_xmmreglo2 001:F9:mod xmmreg1 r/m
VPSUBD — Subtract Packed Integers	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:FA:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:FA:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:FA:11 xmmreg1 xmmreglo3

Instruction and Format	Encoding
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:FA:mod xmmreg1 r/m
VPSUBQ — Subtract Packed Quadword Integers	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:FB:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:FB:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:FB:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:FB:mod xmmreg1 r/m
VPSUBSB — Subtract Packed Signed Integers with Signed Saturation	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:E8:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:E8:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:E8:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:E8:mod xmmreg1 r/m
VPSUBSW — Subtract Packed Signed Integers with Signed Saturation	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:E9:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:E9:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:E9:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:E9:mod xmmreg1 r/m
VPSUBUSB — Subtract Packed Unsigned Integers with Unsigned Saturation	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:D8:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:D8:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:D8:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:D8:mod xmmreg1 r/m
VPSUBUSW — Subtract Packed Unsigned Integers with Unsigned Saturation	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:D9:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:D9:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:D9:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:D9:mod xmmreg1 r/m
VPUNPCKHBW — Unpack High Data	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:68:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:68:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:68:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:68:mod xmmreg1 r/m
VPUNPCKHWD — Unpack High Data	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:69:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:69:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:69:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:69:mod xmmreg1 r/m
VPUNPCKHDQ — Unpack High Data	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:6A:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:6A:mod xmmreg1 r/m

Instruction and Format	Encoding
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:6A:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:6A:mod xmmreg1 r/m
VPUNPCKHQDQ — Unpack High Data	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:6D:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:6D:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:6D:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:6D:mod xmmreg1 r/m
VPUNPCKLBW — Unpack Low Data	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:60:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:60:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:60:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:60:mod xmmreg1 r/m
VPUNPCKLWD — Unpack Low Data	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:61:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:61:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:61:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:61:mod xmmreg1 r/m
VPUNPCKLDQ — Unpack Low Data	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:62:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:62:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:62:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:62:mod xmmreg1 r/m
VPUNPCKLQDQ — Unpack Low Data	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:6C:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:6C:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:6C:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:6C:mod xmmreg1 r/m
VPXOR — Logical Exclusive OR	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:EF:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:EF:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:EF:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:EF:mod xmmreg1 r/m
VSHUFPD — Shuffle Packed Double Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1 using imm8	C4: rxb0_1: w xmmreg2 001:C6:11 xmmreg1 xmmreg3: imm
xmmreg2 with mem to xmmreg1 using imm8	C4: rxb0_1: w xmmreg2 001:C6:mod xmmreg1 r/m: imm
xmmreglo2 with xmmreglo3 to xmmreg1 using imm8	C5: r_xmmreglo2 001:C6:11 xmmreg1 xmmreglo3: imm
xmmreglo2 with mem to xmmreg1 using imm8	C5: r_xmmreglo2 001:C6:mod xmmreg1 r/m: imm
ymmreg2 with ymmreg3 to ymmreg1 using imm8	C4: rxb0_1: w ymmreg2 101:C6:11 ymmreg1 ymmreg3: imm
ymmreg2 with mem to ymmreg1 using imm8	C4: rxb0_1: w ymmreg2 101:C6:mod ymmreg1 r/m: imm

Instruction and Format	Encoding
ymmreglo2 with ymmreglo3 to ymmreg1 using imm8	C5: r_ymmreglo2 101:C6:11 ymmreg1 ymmreglo3: imm
ymmreglo2 with mem to ymmreg1 using imm8	C5: r_ymmreglo2 101:C6:mod ymmreg1 r/m: imm
VSQRTPD — Compute Square Roots of Packed Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 001:51:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 001:51:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 001:51:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 001:51:mod xmmreg1 r/m
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 101:51:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 101:51:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 101:51:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 101:51:mod ymmreg1 r/m
VSQRTSD — Compute Square Root of Scalar Double Precision Floating-Point Value	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:51:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 011:51:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:51:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 011:51:mod xmmreg1 r/m
VSUBPD — Subtract Packed Double Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:5C:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:5C:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:5C:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:5C:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:5C:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:5C:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:5C:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:5C:mod ymmreg1 r/m
VSUBSD — Subtract Scalar Double Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:5C:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 011:5C:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:5C:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 011:5C:mod xmmreg1 r/m
VUCOMISD — Unordered Compare Scalar Double Precision Floating-Point Values and Set EFLAGS	
xmmreg2 with xmmreg1, set EFLAGS	C4: rxb0_1: w_F xmmreg1 001:2E:11 xmmreg2
mem with xmmreg1, set EFLAGS	C4: rxb0_1: w_F xmmreg1 001:2E:mod r/m
xmmreglo with xmmreg1, set EFLAGS	C5: r_F xmmreg1 001:2E:11 xmmreglo
mem with xmmreg1, set EFLAGS	C5: r_F xmmreg1 001:2E:mod r/m
VUNPCKHPD — Unpack and Interleave High Packed Double Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:15:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:15:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:15:11 xmmreg1 xmmreglo3

Instruction and Format	Encoding
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:15:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:15:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:15:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:15:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:15:mod ymmreg1 r/m
VUNPCKHPS — Unpack and Interleave High Packed Single Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:15:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:15:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:15:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:15:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:15:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:15:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 100:15:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 100:15:mod ymmreg1 r/m
VUNPCKLPD — Unpack and Interleave Low Packed Double Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:14:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:14:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:14:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:14:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:14:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:14:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:14:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:14:mod ymmreg1 r/m
VUNPCKLPS — Unpack and Interleave Low Packed Single Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:14:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:14:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:14:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:14:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:14:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:14:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 100:14:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 100:14:mod ymmreg1 r/m
VXORPD — Bitwise Logical XOR for Double Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:57:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:57:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:57:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:57:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:57:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:57:mod ymmreg1 r/m

Instruction and Format	Encoding
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:57:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:57:mod ymmreg1 r/m
VADDPS — Add Packed Single Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:58:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:58:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:58:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:58:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:58:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:58:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 100:58:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 100:58:mod ymmreg1 r/m
VADDSS — Add Scalar Single Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:58:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 010:58:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 010:58:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 010:58:mod xmmreg1 r/m
VANDPS — Bitwise Logical AND of Packed Single Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:54:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:54:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:54:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:54:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:54:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:54:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 100:54:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 100:54:mod ymmreg1 r/m
VANDNPS — Bitwise Logical AND NOT of Packed Single Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:55:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:55:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:55:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:55:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:55:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:55:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 100:55:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 100:55:mod ymmreg1 r/m
VCMPPS — Compare Packed Single Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:C2:11 xmmreg1 xmmreg3: imm
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:C2:mod xmmreg1 r/m: imm
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:C2:11 xmmreg1 xmmreglo3: imm
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:C2:mod xmmreg1 r/m: imm

Instruction and Format	Encoding
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:C2:11 ymmreg1 ymmreg3: imm
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:C2:mod ymmreg1 r/m: imm
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 100:C2:11 ymmreg1 ymmreglo3: imm
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 100:C2:mod ymmreg1 r/m: imm
VCMPSS — Compare Scalar Single Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:C2:11 xmmreg1 xmmreg3: imm
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 010:C2:mod xmmreg1 r/m: imm
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 010:C2:11 xmmreg1 xmmreglo3: imm
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 010:C2:mod xmmreg1 r/m: imm
VCOMISS — Compare Scalar Ordered Single Precision Floating-Point Values and Set EFLAGS	
xmmreg2 with xmmreg1	C4: rxb0_1: w_F 000:2F:11 xmmreg1 xmmreg2
mem with xmmreg1	C4: rxb0_1: w_F 000:2F:mod xmmreg1 r/m
xmmreglo with xmmreg1	C5: r_F 000:2F:11 xmmreg1 xmmreglo
mem with xmmreg1	C5: r_F 000:2F:mod xmmreg1 r/m
VCVTSI2SS — Convert Signed Integer to Scalar Single Precision FP Value	
xmmreg2 with reg to xmmreg1	C4: rxb0_1: 0 xmmreg2 010:2A:11 xmmreg1 reg
xmmreg2 with mem to xmmreg1	C4: rxb0_1: 0 xmmreg2 010:2A:mod xmmreg1 r/m
xmmreglo2 with reglo to xmmreg1	C5: r_xmmreglo2 010:2A:11 xmmreg1 reglo
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 010:2A:mod xmmreg1 r/m
xmmreg2 with reg to xmmreg1	C4: rxb0_1: 1 xmmreg2 010:2A:11 xmmreg1 reg
xmmreg2 with mem to xmmreg1	C4: rxb0_1: 1 xmmreg2 010:2A:mod xmmreg1 r/m
VCVTSS2SI — Convert Scalar Single Precision FP Value to Signed Integer	
xmmreg1 to reg	C4: rxb0_1: 0_F 010:2D:11 reg xmmreg1
mem to reg	C4: rxb0_1: 0_F 010:2D:mod reg r/m
xmmreglo to reg	C5: r_F 010:2D:11 reg xmmreglo
mem to reg	C5: r_F 010:2D:mod reg r/m
xmmreg1 to reg	C4: rxb0_1: 1_F 010:2D:11 reg xmmreg1
mem to reg	C4: rxb0_1: 1_F 010:2D:mod reg r/m
VCVTTSS2SI — Convert with Truncation Scalar Single Precision FP Value to Signed Integer	
xmmreg1 to reg	C4: rxb0_1: 0_F 010:2C:11 reg xmmreg1
mem to reg	C4: rxb0_1: 0_F 010:2C:mod reg r/m
xmmreglo to reg	C5: r_F 010:2C:11 reg xmmreglo
mem to reg	C5: r_F 010:2C:mod reg r/m
xmmreg1 to reg	C4: rxb0_1: 1_F 010:2C:11 reg xmmreg1
mem to reg	C4: rxb0_1: 1_F 010:2C:mod reg r/m
VDIVPS — Divide Packed Single Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:5E:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:5E:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:5E:11 xmmreg1 xmmreglo3

Instruction and Format	Encoding
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:5E:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:5E:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:5E:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 100:5E:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 100:5E:mod ymmreg1 r/m
VDIVSS — Divide Scalar Single Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:5E:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 010:5E:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 010:5E:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 010:5E:mod xmmreg1 r/m
VLDMXCSR — Load MXCSR Register	
mem to MXCSR reg	C4: rxb0_1: w_F 000:AEmod 011 r/m
mem to MXCSR reg	C5: r_F 000:AEmod 011 r/m
VMAXPS — Return Maximum Packed Single Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:5F:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:5F:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:5F:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:5F:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:5F:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:5F:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 100:5F:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 100:5F:mod ymmreg1 r/m
VMAXSS — Return Maximum Scalar Single Precision Floating-Point Value	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:5F:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 010:5F:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 010:5F:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 010:5F:mod xmmreg1 r/m
VMINPS — Return Minimum Packed Single Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:5D:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:5D:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:5D:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:5D:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:5D:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:5D:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 100:5D:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 100:5D:mod ymmreg1 r/m
VMINSS — Return Minimum Scalar Single Precision Floating-Point Value	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:5D:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 010:5D:mod xmmreg1 r/m

Instruction and Format	Encoding
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 010:5D:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 010:5D:mod xmmreg1 r/m
VMOVAPS— Move Aligned Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 000:28:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 000:28:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 000:28:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 000:28:mod xmmreg1 r/m
xmmreg1 to xmmreg2	C4: rxb0_1: w_F 000:29:11 xmmreg2 xmmreg1
xmmreg1 to mem	C4: rxb0_1: w_F 000:29:mod r/m xmmreg1
xmmreg1 to xmmreglo	C5: r_F 000:29:11 xmmreglo xmmreg1
xmmreg1 to mem	C5: r_F 000:29:mod r/m xmmreg1
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 100:28:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 100:28:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 100:28:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 100:28:mod ymmreg1 r/m
ymmreg1 to ymmreg2	C4: rxb0_1: w_F 100:29:11 ymmreg2 ymmreg1
ymmreg1 to mem	C4: rxb0_1: w_F 100:29:mod r/m ymmreg1
ymmreg1 to ymmreglo	C5: r_F 100:29:11 ymmreglo ymmreg1
ymmreg1 to mem	C5: r_F 100:29:mod r/m ymmreg1
VMOVHPS — Move High Packed Single Precision Floating-Point Values	
xmmreg1 with mem to xmmreg2	C4: rxb0_1: w xmmreg1 000:16:mod xmmreg2 r/m
xmmreg1 with mem to xmmreglo2	C5: r_xmmreg1 000:16:mod xmmreglo2 r/m
xmmreg1 to mem	C4: rxb0_1: w_F 000:17:mod r/m xmmreg1
xmmreglo to mem	C5: r_F 000:17:mod r/m xmmreglo
VMOVLHPS — Move Packed Single Precision Floating-Point Values Low to High	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:16:11 xmmreg1 xmmreg3
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:16:11 xmmreg1 xmmreglo3
VMOVLPS — Move Low Packed Single Precision Floating-Point Values	
xmmreg1 with mem to xmmreg2	C4: rxb0_1: w xmmreg1 000:12:mod xmmreg2 r/m
xmmreg1 with mem to xmmreglo2	C5: r_xmmreg1 000:12:mod xmmreglo2 r/m
xmmreg1 to mem	C4: rxb0_1: w_F 000:13:mod r/m xmmreg1
xmmreglo to mem	C5: r_F 000:13:mod r/m xmmreglo
VMOVMSKPS — Extract Packed Single Precision Floating-Point Sign Mask	
xmmreg2 to reg	C4: rxb0_1: w_F 000:50:11 reg xmmreg2
xmmreglo to reg	C5: r_F 000:50:11 reg xmmreglo
ymmreg2 to reg	C4: rxb0_1: w_F 100:50:11 reg ymmreg2
ymmreglo to reg	C5: r_F 100:50:11 reg ymmreglo
VMOVNTPS — Store Packed Single Precision Floating-Point Values Using Non-Temporal Hint	
xmmreg1 to mem	C4: rxb0_1: w_F 000:2B:mod r/m xmmreg1

Instruction and Format	Encoding
xmmreglo to mem	C5: r_F 000:2B:mod r/m xmmreglo
ymmreg1 to mem	C4: rxb0_1: w_F 100:2B:mod r/m ymmreg1
ymmreglo to mem	C5: r_F 100:2B:mod r/m ymmreglo
VMOVSS — Move Scalar Single Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:10:11 xmmreg1 xmmreg3
mem to xmmreg1	C4: rxb0_1: w_F 010:10:mod xmmreg1 r/m
xmmreg2 with xmmreg3 to xmmreg1	C5: r_xmmreg2 010:10:11 xmmreg1 xmmreg3
mem to xmmreg1	C5: r_F 010:10:mod xmmreg1 r/m
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:11:11 xmmreg1 xmmreg3
xmmreg1 to mem	C4: rxb0_1: w_F 010:11:mod r/m xmmreg1
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 010:11:11 xmmreg1 xmmreglo3
xmmreglo to mem	C5: r_F 010:11:mod r/m xmmreglo
VMOVUPS— Move Unaligned Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 000:10:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 000:10:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 000:10:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 000:10:mod xmmreg1 r/m
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 100:10:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 100:10:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 100:10:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 100:10:mod ymmreg1 r/m
xmmreg1 to xmmreg2	C4: rxb0_1: w_F 000:11:11 xmmreg2 xmmreg1
xmmreg1 to mem	C4: rxb0_1: w_F 000:11:mod r/m xmmreg1
xmmreg1 to xmmreglo	C5: r_F 000:11:11 xmmreglo xmmreg1
xmmreg1 to mem	C5: r_F 000:11:mod r/m xmmreg1
ymmreg1 to ymmreg2	C4: rxb0_1: w_F 100:11:11 ymmreg2 ymmreg1
ymmreg1 to mem	C4: rxb0_1: w_F 100:11:mod r/m ymmreg1
ymmreg1 to ymmreglo	C5: r_F 100:11:11 ymmreglo ymmreg1
ymmreg1 to mem	C5: r_F 100:11:mod r/m ymmreg1
VMULPS — Multiply Packed Single Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:59:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:59:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:59:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:59:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:59:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:59:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 100:59:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 100:59:mod ymmreg1 r/m
VMULSS — Multiply Scalar Single Precision Floating-Point Values	

Instruction and Format	Encoding
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:59:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 010:59:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 010:59:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 010:59:mod xmmreg1 r/m
VORPS — Bitwise Logical OR of Single Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:56:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:56:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:56:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:56:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:56:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:56:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 100:56:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 100:56:mod ymmreg1 r/m
VRCPPS — Compute Reciprocals of Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 000:53:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 000:53:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 000:53:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 000:53:mod xmmreg1 r/m
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 100:53:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 100:53:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 100:53:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 100:53:mod ymmreg1 r/m
VRCPSS — Compute Reciprocal of Scalar Single Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:53:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 010:53:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 010:53:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 010:53:mod xmmreg1 r/m
VRSQRTPS — Compute Reciprocals of Square Roots of Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 000:52:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 000:52:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 000:52:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 000:52:mod xmmreg1 r/m
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 100:52:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 100:52:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 100:52:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 100:52:mod ymmreg1 r/m
VRSQRTSS — Compute Reciprocal of Square Root of Scalar Single Precision Floating-Point Value	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:52:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 010:52:mod xmmreg1 r/m

Instruction and Format	Encoding
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 010:52:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 010:52:mod xmmreg1 r/m
VSHUFPS — Shuffle Packed Single Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1, imm8	C4: rxb0_1: w xmmreg2 000:C6:11 xmmreg1 xmmreg3: imm
xmmreg2 with mem to xmmreg1, imm8	C4: rxb0_1: w xmmreg2 000:C6:mod xmmreg1 r/m: imm
xmmreglo2 with xmmreglo3 to xmmreg1, imm8	C5: r_xmmreglo2 000:C6:11 xmmreg1 xmmreglo3: imm
xmmreglo2 with mem to xmmreg1, imm8	C5: r_xmmreglo2 000:C6:mod xmmreg1 r/m: imm
ymmreg2 with ymmreg3 to ymmreg1, imm8	C4: rxb0_1: w ymmreg2 100:C6:11 ymmreg1 ymmreg3: imm
ymmreg2 with mem to ymmreg1, imm8	C4: rxb0_1: w ymmreg2 100:C6:mod ymmreg1 r/m: imm
ymmreglo2 with ymmreglo3 to ymmreg1, imm8	C5: r_ymmreglo2 100:C6:11 ymmreg1 ymmreglo3: imm
ymmreglo2 with mem to ymmreg1, imm8	C5: r_ymmreglo2 100:C6:mod ymmreg1 r/m: imm
VSQRTPS — Compute Square Roots of Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 000:51:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 000:51:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 000:51:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 000:51:mod xmmreg1 r/m
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 100:51:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 100:51:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 100:51:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 100:51:mod ymmreg1 r/m
VSQRTSS — Compute Square Root of Scalar Single Precision Floating-Point Value	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:51:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 010:51:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 010:51:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 010:51:mod xmmreg1 r/m
VSTMXCSR — Store MXCSR Register State	
MXCSR to mem	C4: rxb0_1: w_F 000:AE:mod 011 r/m
MXCSR to mem	C5: r_F 000:AE:mod 011 r/m
VSUBPS — Subtract Packed Single Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:5C:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:5C:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:5C:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:5C:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:5C:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:5C:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 100:5C:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 100:5C:mod ymmreg1 r/m
VSUBSS — Subtract Scalar Single Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:5C:11 xmmreg1 xmmreg3

Instruction and Format	Encoding
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 010:5C:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 010:5C:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 010:5C:mod xmmreg1 r/m
VUCOMISS — Unordered Compare Scalar Single Precision Floating-Point Values and Set EFLAGS	
xmmreg2 with xmmreg1	C4: rxb0_1: w_F 000:2E:11 xmmreg1 xmmreg2
mem with xmmreg1	C4: rxb0_1: w_F 000:2E:mod xmmreg1 r/m
xmmreglo with xmmreg1	C5: r_F 000:2E:11 xmmreg1 xmmreglo
mem with xmmreg1	C5: r_F 000:2E:mod xmmreg1 r/m
UNPCKHPS — Unpack and Interleave High Packed Single Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:15:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:15mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:15:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:15mod ymmreg1 r/m
UNPCKLPS — Unpack and Interleave Low Packed Single Precision Floating-Point Value	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:14:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:14mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:14:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:14mod ymmreg1 r/m
VXORPS — Bitwise Logical XOR for Single Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:57:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:57:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:57:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:57:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:57:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:57:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 100:57:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 100:57:mod ymmreg1 r/m
VBROADCAST —Load with Broadcast	
mem to xmmreg1	C4: rxb0_2: 0_F 001:18:mod xmmreg1 r/m
mem to ymmreg1	C4: rxb0_2: 0_F 101:18:mod ymmreg1 r/m
mem to ymmreg1	C4: rxb0_2: 0_F 101:19:mod ymmreg1 r/m
mem to ymmreg1	C4: rxb0_2: 0_F 101:1A:mod ymmreg1 r/m
VEXTRACTF128 — Extract Packed Floating-Point Values	
ymmreg2 to xmmreg1, imm8	C4: rxb0_3: 0_F 001:19:11 xmmreg1 ymmreg2: imm
ymmreg2 to mem, imm8	C4: rxb0_3: 0_F 001:19:mod r/m ymmreg2: imm
VINSERTF128 — Insert Packed Floating-Point Values	
xmmreg3 and merge with ymmreg2 to ymmreg1, imm8	C4: rxb0_3: 0 ymmreg2101:18:11 ymmreg1 xmmreg3: imm
mem and merge with ymmreg2 to ymmreg1, imm8	C4: rxb0_3: 0 ymmreg2 101:18:mod ymmreg1 r/m: imm
VPERMILPD — Permute Double Precision Floating-Point Values	

Instruction and Format	Encoding
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: 0 xmmreg2 001:0D:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: 0 xmmreg2 001:0D:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_2: 0 ymmreg2 101:0D:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_2: 0 ymmreg2 101:0D:mod ymmreg1 r/m
xmmreg2 to xmmreg1, imm	C4: rxb0_3: 0_F 001:05:11 xmmreg1 xmmreg2: imm
mem to xmmreg1, imm	C4: rxb0_3: 0_F 001:05:mod xmmreg1 r/m: imm
ymmreg2 to ymmreg1, imm	C4: rxb0_3: 0_F 101:05:11 ymmreg1 ymmreg2: imm
mem to ymmreg1, imm	C4: rxb0_3: 0_F 101:05:mod ymmreg1 r/m: imm
VPERMILPS — Permute Single Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: 0 xmmreg2 001:0C:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: 0 xmmreg2 001:0C:mod xmmreg1 r/m
xmmreg2 to xmmreg1, imm	C4: rxb0_3: 0_F 001:04:11 xmmreg1 xmmreg2: imm
mem to xmmreg1, imm	C4: rxb0_3: 0_F 001:04:mod xmmreg1 r/m: imm
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_2: 0 ymmreg2 101:0C:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_2: 0 ymmreg2 101:0C:mod ymmreg1 r/m
ymmreg2 to ymmreg1, imm	C4: rxb0_3: 0_F 101:04:11 ymmreg1 ymmreg2: imm
mem to ymmreg1, imm	C4: rxb0_3: 0_F 101:04:mod ymmreg1 r/m: imm
VPERM2F128 — Permute Floating-Point Values	
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_3: 0 ymmreg2 101:06:11 ymmreg1 ymmreg3: imm
ymmreg2 with mem to ymmreg1	C4: rxb0_3: 0 ymmreg2 101:06:mod ymmreg1 r/m: imm
VTESTPD/VTESTPS — Packed Bit Test	
xmmreg2 to xmmreg1	C4: rxb0_2: 0_F 001:0E:11 xmmreg2 xmmreg1
mem to xmmreg1	C4: rxb0_2: 0_F 001:0E:mod xmmreg2 r/m
ymmreg2 to ymmreg1	C4: rxb0_2: 0_F 101:0E:11 ymmreg2 ymmreg1
mem to ymmreg1	C4: rxb0_2: 0_F 101:0E:mod ymmreg2 r/m
xmmreg2 to xmmreg1	C4: rxb0_2: 0_F 001:0F:11 xmmreg1 xmmreg2: imm
mem to xmmreg1	C4: rxb0_2: 0_F 001:0F:mod xmmreg1 r/m: imm
ymmreg2 to ymmreg1	C4: rxb0_2: 0_F 101:0F:11 ymmreg1 ymmreg2: imm
mem to ymmreg1	C4: rxb0_2: 0_F 101:0F:mod ymmreg1 r/m: imm

NOTES:

1. The term "lo" refers to the lower eight registers, 0-7

B.17 FLOATING-POINT INSTRUCTION FORMATS AND ENCODINGS

Table B-38 shows the five different formats used for floating-point instructions. In all cases, instructions are at least two bytes long and begin with the bit pattern 11011.

Table B-38. General Floating-Point Instruction Formats

	Instruction										Optional Fields	
	First Byte			Second Byte								
1	11011	OPA		1	mod		1	OPB		r/m	s-i-b	disp
2	11011	MF		OPA	mod		OPB			r/m	s-i-b	disp
3	11011	d	P	OPA	1	1	OPB	R		ST(i)		
4	11011	0	0	1	1	1	1	OP				
5	11011	0	1	1	1	1	1	OP				
	15–11	10	9	8	7	6	5	4	3	2 1 0		

MF = Memory Format
- 00 — 32-bit real
- 01 — 32-bit integer
- 10 — 64-bit real
- 11 — 16-bit integer

P = Pop
- 0 — Do not pop stack
- 1 — Pop stack after operation

d = Destination
- 0 — Destination is ST(0)
- 1 — Destination is ST(i)

R XOR d = 0 — Destination OP Source
R XOR d = 1 — Source OP Destination

ST(i) = Register stack element *i*
- 000 = Stack Top
- 001 = Second stack element
- .
- .
- .
- 111 = Eighth stack element

The Mod and R/M fields of the ModR/M byte have the same interpretation as the corresponding fields of the integer instructions. The SIB byte and disp (displacement) are optionally present in instructions that have Mod and R/M fields. Their presence depends on the values of Mod and R/M, as for integer instructions.

Table B-39 shows the formats and encodings of the floating-point instructions.

Table B-39. Floating-Point Instruction Formats and Encodings

Instruction and Format	Encoding
F2XM1 – Compute $2^{ST(0)} - 1$	11011 001 : 1111 0000
FABS – Absolute Value	11011 001 : 1110 0001
FADD – Add	
ST(0) := ST(0) + 32-bit memory	11011 000 : mod 000 r/m
ST(0) := ST(0) + 64-bit memory	11011 100 : mod 000 r/m
ST(d) := ST(0) + ST(i)	11011 d00 : 11 000 ST(i)
FADDP – Add and Pop	
ST(0) := ST(0) + ST(i)	11011 110 : 11 000 ST(i)
FBLD – Load Binary Coded Decimal	11011 111 : mod 100 r/m
FBSTP – Store Binary Coded Decimal and Pop	11011 111 : mod 110 r/m
FCHS – Change Sign	11011 001 : 1110 0000
FCLEX – Clear Exceptions	11011 011 : 1110 0010
FCOM – Compare Real	

Table B-39. Floating-Point Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
32-bit memory	11011 000 : mod 010 r/m
64-bit memory	11011 100 : mod 010 r/m
ST(i)	11011 000 : 11 010 ST(i)
FCOMP – Compare Real and Pop	
32-bit memory	11011 000 : mod 011 r/m
64-bit memory	11011 100 : mod 011 r/m
ST(i)	11011 000 : 11 011 ST(i)
FCOMPP – Compare Real and Pop Twice	11011 110 : 11 011 001
FCOMIP – Compare Real, Set EFLAGS, and Pop	11011 111 : 11 110 ST(i)
FCOS – Cosine of ST(0)	11011 001 : 1111 1111
FDECSTP – Decrement Stack-Top Pointer	11011 001 : 1111 0110
FDIV – Divide	
ST(0) := ST(0) ÷ 32-bit memory	11011 000 : mod 110 r/m
ST(0) := ST(0) ÷ 64-bit memory	11011 100 : mod 110 r/m
ST(d) := ST(0) ÷ ST(i)	11011 d00 : 1111 R ST(i)
FDIVP – Divide and Pop	
ST(0) := ST(0) ÷ ST(i)	11011 110 : 1111 1 ST(i)
FDIVR – Reverse Divide	
ST(0) := 32-bit memory ÷ ST(0)	11011 000 : mod 111 r/m
ST(0) := 64-bit memory ÷ ST(0)	11011 100 : mod 111 r/m
ST(d) := ST(i) ÷ ST(0)	11011 d00 : 1111 R ST(i)
FDIVRP – Reverse Divide and Pop	
ST(0) := ST(i) ÷ ST(0)	11011 110 : 1111 0 ST(i)
FFREE – Free ST(i) Register	11011 101 : 1100 0 ST(i)
FIADD – Add Integer	
ST(0) := ST(0) + 16-bit memory	11011 110 : mod 000 r/m
ST(0) := ST(0) + 32-bit memory	11011 010 : mod 000 r/m
FICOM – Compare Integer	
16-bit memory	11011 110 : mod 010 r/m
32-bit memory	11011 010 : mod 010 r/m
FICOMP – Compare Integer and Pop	
16-bit memory	11011 110 : mod 011 r/m
32-bit memory	11011 010 : mod 011 r/m
FIDIV – Divide	
ST(0) := ST(0) ÷ 16-bit memory	11011 110 : mod 110 r/m
ST(0) := ST(0) ÷ 32-bit memory	11011 010 : mod 110 r/m
FIDIVR – Reverse Divide	
ST(0) := 16-bit memory ÷ ST(0)	11011 110 : mod 111 r/m

Table B-39. Floating-Point Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
ST(0) := 32-bit memory ÷ ST(0)	11011 010 : mod 111 r/m
FILD – Load Integer	
16-bit memory	11011 111 : mod 000 r/m
32-bit memory	11011 011 : mod 000 r/m
64-bit memory	11011 111 : mod 101 r/m
FIMUL– Multiply	
ST(0) := ST(0) × 16-bit memory	11011 110 : mod 001 r/m
ST(0) := ST(0) × 32-bit memory	11011 010 : mod 001 r/m
FINCSTP – Increment Stack Pointer	11011 001 : 1111 0111
FINIT – Initialize Floating-Point Unit	
FIST – Store Integer	
16-bit memory	11011 111 : mod 010 r/m
32-bit memory	11011 011 : mod 010 r/m
FISTP – Store Integer and Pop	
16-bit memory	11011 111 : mod 011 r/m
32-bit memory	11011 011 : mod 011 r/m
64-bit memory	11011 111 : mod 111 r/m
FISUB – Subtract	
ST(0) := ST(0) - 16-bit memory	11011 110 : mod 100 r/m
ST(0) := ST(0) - 32-bit memory	11011 010 : mod 100 r/m
FISUBR – Reverse Subtract	
ST(0) := 16-bit memory – ST(0)	11011 110 : mod 101 r/m
ST(0) := 32-bit memory – ST(0)	11011 010 : mod 101 r/m
FLD – Load Real	
32-bit memory	11011 001 : mod 000 r/m
64-bit memory	11011 101 : mod 000 r/m
80-bit memory	11011 011 : mod 101 r/m
ST(i)	11011 001 : 11 000 ST(i)
FLD1 – Load +1.0 into ST(0)	11011 001 : 1110 1000
FLDCW – Load Control Word	11011 001 : mod 101 r/m
FLDENV – Load FPU Environment	11011 001 : mod 100 r/m
FLDL2E – Load $\log_2(\varepsilon)$ into ST(0)	11011 001 : 1110 1010
FLDL2T – Load $\log_2(10)$ into ST(0)	11011 001 : 1110 1001
FLDLG2 – Load $\log_{10}(2)$ into ST(0)	11011 001 : 1110 1100
FLDLN2 – Load $\log_{\varepsilon}(2)$ into ST(0)	11011 001 : 1110 1101
FLDPI – Load π into ST(0)	11011 001 : 1110 1011
FLDZ – Load +0.0 into ST(0)	11011 001 : 1110 1110
FMUL – Multiply	

Table B-39. Floating-Point Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
ST(0) := ST(0) × 32-bit memory	11011 000 : mod 001 r/m
ST(0) := ST(0) × 64-bit memory	11011 100 : mod 001 r/m
ST(d) := ST(0) × ST(i)	11011 d00 : 1100 1 ST(i)
FMULP – Multiply	
ST(i) := ST(0) × ST(i)	11011 110 : 1100 1 ST(i)
FNOP – No Operation	11011 001 : 1101 0000
FPATAN – Partial Arctangent	11011 001 : 1111 0011
FPREM – Partial Remainder	11011 001 : 1111 1000
FPREM1 – Partial Remainder (IEEE)	11011 001 : 1111 0101
FPTAN – Partial Tangent	11011 001 : 1111 0010
FRNDINT – Round to Integer	11011 001 : 1111 1100
FRSTOR – Restore FPU State	11011 101 : mod 100 r/m
FSAVE – Store FPU State	11011 101 : mod 110 r/m
FSCALE – Scale	11011 001 : 1111 1101
FSIN – Sine	11011 001 : 1111 1110
FSINCOS – Sine and Cosine	11011 001 : 1111 1011
FSQRT – Square Root	11011 001 : 1111 1010
FST – Store Real	
32-bit memory	11011 001 : mod 010 r/m
64-bit memory	11011 101 : mod 010 r/m
ST(i)	11011 101 : 11 010 ST(i)
FSTCW – Store Control Word	11011 001 : mod 111 r/m
FSTENV – Store FPU Environment	11011 001 : mod 110 r/m
FSTP – Store Real and Pop	
32-bit memory	11011 001 : mod 011 r/m
64-bit memory	11011 101 : mod 011 r/m
80-bit memory	11011 011 : mod 111 r/m
ST(i)	11011 101 : 11 011 ST(i)
FSTSW – Store Status Word into AX	11011 111 : 1110 0000
FSTSW – Store Status Word into Memory	11011 101 : mod 111 r/m
FSUB – Subtract	
ST(0) := ST(0) – 32-bit memory	11011 000 : mod 100 r/m
ST(0) := ST(0) – 64-bit memory	11011 100 : mod 100 r/m
ST(d) := ST(0) – ST(i)	11011 d00 : 1110 R ST(i)
FSUBP – Subtract and Pop	
ST(0) := ST(0) – ST(i)	11011 110 : 1110 1 ST(i)
FSUBR – Reverse Subtract	
ST(0) := 32-bit memory – ST(0)	11011 000 : mod 101 r/m

Table B-39. Floating-Point Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
ST(0) := 64-bit memory – ST(0)	11011 100 : mod 101 r/m
ST(d) := ST(i) – ST(0)	11011 d00 : 1110 R ST(i)
FSUBRP – Reverse Subtract and Pop	
ST(i) := ST(i) – ST(0)	11011 110 : 1110 0 ST(i)
FTST – Test	11011 001 : 1110 0100
FUCOM – Unordered Compare Real	11011 101 : 1110 0 ST(i)
FUCOMP – Unordered Compare Real and Pop	11011 101 : 1110 1 ST(i)
FUCOMPP – Unordered Compare Real and Pop Twice	11011 010 : 1110 1001
FUCOMI – Unorderd Compare Real and Set EFLAGS	11011 011 : 11 101 ST(i)
FUCOMIP – Unorderd Compare Real, Set EFLAGS, and Pop	11011 111 : 11 101 ST(i)
FXAM – Examine	11011 001 : 1110 0101
FXCH – Exchange ST(0) and ST(i)	11011 001 : 1100 1 ST(i)
FXTRACT – Extract Exponent and Significand	11011 001 : 1111 0100
FYL2X – ST(1) $\times$ $\log_2$(ST(0))	11011 001 : 1111 0001
FYL2XP1 – ST(1) $\times$ $\log_2$(ST(0) + 1.0)	11011 001 : 1111 1001
FWAIT – Wait until FPU Ready	1001 1011 (same instruction as WAIT)

B.18 VMX INSTRUCTIONS

Table B-40 describes virtual-machine extensions (VMX).

Table B-40. Encodings for VMX Instructions

Instruction and Format	Encoding
INVEPT—Invalidate Cached EPT Mappings	
Descriptor m128 according to reg	01100110 00001111 00111000 10000000: mod reg r/m
INVVPID—Invalidate Cached VPID Mappings	
Descriptor m128 according to reg	01100110 00001111 00111000 10000001: mod reg r/m
VMCALL—Call to VM Monitor	
Call VMM: causes VM exit	00001111 00000001 11000001
VMCLEAR—Clear Virtual-Machine Control Structure	
mem32:VMCS_data_ptr	01100110 00001111 11000111: mod 110 r/m
mem64:VMCS_data_ptr	01100110 00001111 11000111: mod 110 r/m
VMFUNC—Invoke VM Function	
Invoke VM function specified in EAX	00001111 00000001 11010100
VMLAUNCH—Launch Virtual Machine	
Launch VM managed by Current_VMCS	00001111 00000001 11000010
VMRESUME—Resume Virtual Machine	
Resume VM managed by Current_VMCS	00001111 00000001 11000011
VMPTRLD—Load Pointer to Virtual-Machine Control Structure	
mem32 to Current_VMCS_ptr	00001111 11000111: mod 110 r/m

Table B-40. Encodings for VMX Instructions

Instruction and Format	Encoding
mem64 to Current_VMCS_ptr	00001111 11000111: mod 110 r/m
VMPTRST—Store Pointer to Virtual-Machine Control Structure	
Current_VMCS_ptr to mem32	00001111 11000111: mod 111 r/m
Current_VMCS_ptr to mem64	00001111 11000111: mod 111 r/m
VMREAD—Read Field from Virtual-Machine Control Structure	
r32 *(VMCS_fieldn)* to r32	00001111 01111000: 11 reg2 reg1
r32 *(VMCS_fieldn)* to mem32	00001111 01111000: mod r32 r/m
r64 *(VMCS_fieldn)* to r64	00001111 01111000: 11 reg2 reg1
r64 *(VMCS_fieldn)* to mem64	00001111 01111000: mod r64 r/m
VMWRITE—Write Field to Virtual-Machine Control Structure	
r32 to r32 *(VMCS_fieldn)*	00001111 01111001: 11 reg1 reg2
mem32 to r32 *(VMCS_fieldn)*	00001111 01111001: mod r32 r/m
r64 to r64 *(VMCS_fieldn)*	00001111 01111001: 11 reg1 reg2
mem64 to r64 *(VMCS_fieldn)*	00001111 01111001: mod r64 r/m
VMXOFF—Leave VMX Operation	
Leave VMX.	00001111 00000001 11000100
VMXON—Enter VMX Operation	
Enter VMX.	11110011 000011111 11000111: mod 110 r/m

B.19 SMX INSTRUCTIONS

Table B-41 describes Safer Mode Extensions (SMX). **GETSEC leaf functions are selected by a valid value in EAX on input.**

Table B-41. Encodings for SMX Instructions

Instruction and Format	Encoding
GETSEC—GETSEC leaf functions are selected by the value in EAX on input	
GETSEC[CAPABILITIES]	00001111 00110111 (EAX= 0)
GETSEC[ENTERACCS]	00001111 00110111 (EAX= 2)
GETSEC[EXITAC]	00001111 00110111 (EAX= 3)
GETSEC[SENTER]	00001111 00110111 (EAX= 4)
GETSEC[SEXIT]	00001111 00110111 (EAX= 5)
GETSEC[PARAMETERS]	00001111 00110111 (EAX= 6)
GETSEC[SMCTRL]	00001111 00110111 (EAX= 7)
GETSEC[WAKEUP]	00001111 00110111 (EAX= 8)

APPENDIX C
INTEL® C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

The two tables in this appendix itemize the Intel C/C++ compiler intrinsics and functional equivalents for the Intel MMX technology, SSE, SSE2, SSE3, and SSSE3 instructions.

There may be additional intrinsics that do not have an instruction equivalent. It is strongly recommended that the reader reference the compiler documentation for the complete list of supported intrinsics. Please refer to http://www.intel.com/support/performancetools/.

Table C-1 presents simple intrinsics and Table C-2 presents composite intrinsics. Some intrinsics are "composites" because they require more than one instruction to implement them.

Intel C/C++ Compiler intrinsic names reflect the following naming conventions:

```
_mm_<intrin_op>_<suffix>
```

where:

<intrin_op>	Indicates the intrinsics basic operation; for example, add for addition and sub for subtraction
<suffix>	Denotes the type of data operated on by the instruction. The first one or two letters of each suffix denotes whether the data is packed (p), extended packed (ep), or scalar (s).

The remaining letters denote the type:

s	single precision floating-point
d	double precision floating-point
i128	signed 128-bit integer
i64	signed 64-bit integer
u64	unsigned 64-bit integer
i32	signed 32-bit integer
u32	unsigned 32-bit integer
i16	signed 16-bit integer
u16	unsigned 16-bit integer
i8	signed 8-bit integer
u8	unsigned 8-bit integer

The variable r is generally used for the intrinsic's return value. A number appended to a variable name indicates the element of a packed object. For example, r0 is the lowest word of r.

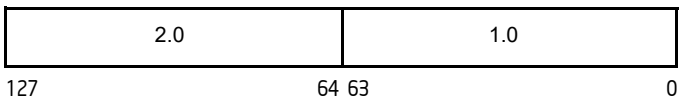
The packed values are represented in right-to-left order, with the lowest value being used for scalar operations. Consider the following example operation:

```
double a[2] = {1.0, 2.0};
__m128d t = _mm_load_pd(a);
```

The result is the same as either of the following:

```
__m128d t = _mm_set_pd(2.0, 1.0);
__m128d t = _mm_setr_pd(1.0, 2.0);
```

In other words, the XMM register that holds the value t will look as follows:



The "scalar" element is 1.0. Due to the nature of the instruction, some intrinsics require their arguments to be immediates (constant integer literals).

To use an intrinsic in your code, insert a line with the following syntax:

```
data_type intrinsic_name (parameters)
```

Where:

data_type	Is the return data type, which can be either void, int, __m64, __m128, __m128d, or __m128i. Only the _mm_empty intrinsic returns void.
intrinsic_name	Is the name of the intrinsic, which behaves like a function that you can use in your C/C++ code instead of in-lining the actual instruction.
parameters	Represents the parameters required by each intrinsic.

C.1 SIMPLE INTRINSICS

NOTE

For detailed descriptions of the intrinsics in Table C-1, see the corresponding mnemonic in Chapter 3, "Instruction Set Reference, A-L," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A; Chapter 4, "Instruction Set Reference, M-U," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B; Chapter 5, "Instruction Set Reference, V," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C; or Chapter 6, "Instruction Set Reference, W-Z," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D.

Table C-1. Simple Intrinsics

Mnemonic	Intrinsic
ADDPD	__m128d _mm_add_pd(__m128d a, __m128d b)
ADDPS	__m128 _mm_add_ps(__m128 a, __m128 b)
ADDSD	__m128d _mm_add_sd(__m128d a, __m128d b)
ADDSS	__m128 _mm_add_ss(__m128 a, __m128 b)
ADDSUBPD	__m128d _mm_addsub_pd(__m128d a, __m128d b)
ADDSUBPS	__m128 _mm_addsub_ps(__m128 a, __m128 b)
AESDEC	__m128i _mm_aesdec (__m128i, __m128i)
AESDECLAST	__m128i _mm_aesdeclast (__m128i, __m128i)
AESENC	__m128i _mm_aesenc (__m128i, __m128i)
AESENCLAST	__m128i _mm_aesenclast (__m128i, __m128i)
AESIMC	__m128i _mm_aesimc (__m128i)
AESKEYGENASSIST	__m128i _mm_aesimc (__m128i, const int)
ANDNPD	__m128d _mm_andnot_pd(__m128d a, __m128d b)
ANDNPS	__m128 _mm_andnot_ps(__m128 a, __m128 b)
ANDPD	__m128d _mm_and_pd(__m128d a, __m128d b)
ANDPS	__m128 _mm_and_ps(__m128 a, __m128 b)
BLENDPD	__m128d _mm_blend_pd(__m128d v1, __m128d v2, const int mask)
BLENDPS	__m128 _mm_blend_ps(__m128 v1, __m128 v2, const int mask)
BLENDVPD	__m128d _mm_blendv_pd(__m128d v1, __m128d v2, __m128d v3)
BLENDVPS	__m128 _mm_blendv_ps(__m128 v1, __m128 v2, __m128 v3)
CLFLUSH	void _mm_clflush(void const *p)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
CMPPD	__m128d _mm_cmpeq_pd(__m128d a, __m128d b)
	__m128d _mm_cmplt_pd(__m128d a, __m128d b)
	__m128d _mm_cmple_pd(__m128d a, __m128d b)
	__m128d _mm_cmpgt_pd(__m128d a, __m128d b)
	__m128d _mm_cmpge_pd(__m128d a, __m128d b)
	__m128d _mm_cmpneq_pd(__m128d a, __m128d b)
	__m128d _mm_cmpnlt_pd(__m128d a, __m128d b)
	__m128d _mm_cmpngt_pd(__m128d a, __m128d b)
	__m128d _mm_cmpnge_pd(__m128d a, __m128d b)
	__m128d _mm_cmpord_pd(__m128d a, __m128d b)
	__m128d _mm_cmpunord_pd(__m128d a, __m128d b)
	__m128d _mm_cmpnle_pd(__m128d a, __m128d b)
CMPPS	__m128 _mm_cmpeq_ps(__m128 a, __m128 b)
	__m128 _mm_cmplt_ps(__m128 a, __m128 b)
	__m128 _mm_cmple_ps(__m128 a, __m128 b)
	__m128 _mm_cmpgt_ps(__m128 a, __m128 b)
	__m128 _mm_cmpge_ps(__m128 a, __m128 b)
	__m128 _mm_cmpneq_ps(__m128 a, __m128 b)
	__m128 _mm_cmpnlt_ps(__m128 a, __m128 b)
	__m128 _mm_cmpngt_ps(__m128 a, __m128 b)
	__m128 _mm_cmpnge_ps(__m128 a, __m128 b)
	__m128 _mm_cmpord_ps(__m128 a, __m128 b)
	__m128 _mm_cmpunord_ps(__m128 a, __m128 b)
	__m128 _mm_cmpnle_ps(__m128 a, __m128 b)
CMPSD	__m128d _mm_cmpeq_sd(__m128d a, __m128d b)
	__m128d _mm_cmplt_sd(__m128d a, __m128d b)
	__m128d _mm_cmple_sd(__m128d a, __m128d b)
	__m128d _mm_cmpgt_sd(__m128d a, __m128d b)
	__m128d _mm_cmpge_sd(__m128d a, __m128d b)
	__m128 _mm_cmpneq_sd(__m128d a, __m128d b)
	__m128 _mm_cmpnlt_sd(__m128d a, __m128d b)
	__m128d _mm_cmpnle_sd(__m128d a, __m128d b)
	__m128d _mm_cmpngt_sd(__m128d a, __m128d b)
	__m128d _mm_cmpnge_sd(__m128d a, __m128d b)
	__m128d _mm_cmpord_sd(__m128d a, __m128d b)
	__m128d _mm_cmpunord_sd(__m128d a, __m128d b)
CMPSS	__m128 _mm_cmpeq_ss(__m128 a, __m128 b)
	__m128 _mm_cmplt_ss(__m128 a, __m128 b)
	__m128 _mm_cmple_ss(__m128 a, __m128 b)
	__m128 _mm_cmpgt_ss(__m128 a, __m128 b)
	__m128 _mm_cmpge_ss(__m128 a, __m128 b)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
	__m128 _mm_cmpneq_ss(__m128 a, __m128 b)
	__m128 _mm_cmpnlt_ss(__m128 a, __m128 b)
	__m128 _mm_cmpnle_ss(__m128 a, __m128 b)
	__m128 _mm_cmpngt_ss(__m128 a, __m128 b)
	__m128 _mm_cmpnge_ss(__m128 a, __m128 b)
	__m128 _mm_cmpord_ss(__m128 a, __m128 b)
	__m128 _mm_cmpunord_ss(__m128 a, __m128 b)
COMISD	int _mm_comieq_sd(__m128d a, __m128d b)
	int _mm_comilt_sd(__m128d a, __m128d b)
	int _mm_comile_sd(__m128d a, __m128d b)
	int _mm_comigt_sd(__m128d a, __m128d b)
	int _mm_comige_sd(__m128d a, __m128d b)
	int _mm_comineq_sd(__m128d a, __m128d b)
COMISS	int _mm_comieq_ss(__m128 a, __m128 b)
	int _mm_comilt_ss(__m128 a, __m128 b)
	int _mm_comile_ss(__m128 a, __m128 b)
	int _mm_comigt_ss(__m128 a, __m128 b)
	int _mm_comige_ss(__m128 a, __m128 b)
	int _mm_comineq_ss(__m128 a, __m128 b)
CRC32	unsigned int _mm_crc32_u8(unsigned int crc, unsigned char data)
	unsigned int _mm_crc32_u16(unsigned int crc, unsigned short data)
	unsigned int _mm_crc32_u32(unsigned int crc, unsigned int data)
	unsigned __int64 _mm_crc32_u64(unsigned __int64 crc, unsigned __int64 data)
CVTDQ2PD	__m128d _mm_cvtepi32_pd(__m128i a)
CVTDQ2PS	__m128 _mm_cvtepi32_ps(__m128i a)
CVTPD2DQ	__m128i _mm_cvtpd_epi32(__m128d a)
CVTPD2PI	__m64 _mm_cvtpd_pi32(__m128d a)
CVTPD2PS	__m128 _mm_cvtpd_ps(__m128d a)
CVTPI2PD	__m128d _mm_cvtpi32_pd(__m64 a)
CVTPI2PS	__m128 _mm_cvt_pi2ps(__m128 a, __m64 b) __m128 _mm_cvtpi32_ps(__m128 a, __m64 b)
CVTPS2DQ	__m128i _mm_cvtps_epi32(__m128 a)
CVTPS2PD	__m128d _mm_cvtps_pd(__m128 a)
CVTPS2PI	__m64 _mm_cvt_ps2pi(__m128 a) __m64 _mm_cvtps_pi32(__m128 a)
CVTSD2SI	int _mm_cvtsd_si32(__m128d a)
CVTSD2SS	__m128 _mm_cvtsd_ss(__m128 a, __m128d b)
CVTSI2SD	__m128d _mm_cvtsi32_sd(__m128d a, int b)
CVTSI2SS	__m128 _mm_cvt_si2ss(__m128 a, int b) __m128 _mm_cvtsi32_ss(__m128 a, int b) __m128 _mm_cvtsi64_ss(__m128 a, __int64 b)
CVTSS2SD	__m128d _mm_cvtss_sd(__m128d a, __m128 b)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
CVTSS2SI	int _mm_cvt_ss2si(__m128 a) int _mm_cvtss_si32(__m128 a)
CVTTPD2DQ	__m128i _mm_cvttpd_epi32(__m128d a)
CVTTPD2PI	__m64 _mm_cvttpd_pi32(__m128d a)
CVTTPS2DQ	__m128i _mm_cvttps_epi32(__m128 a)
CVTTPS2PI	__m64 _mm_cvtt_ps2pi(__m128 a) __m64 _mm_cvttps_pi32(__m128 a)
CVTTSD2SI	int _mm_cvttsd_si32(__m128d a)
CVTTSS2SI	int _mm_cvtt_ss2si(__m128 a) int _mm_cvttss_si32(__m128 a) __m64 _mm_cvtsi32_si64(int i) int _mm_cvtsi64_si32(__m64 m)
DIVPD	__m128d _mm_div_pd(__m128d a, __m128d b)
DIVPS	__m128 _mm_div_ps(__m128 a, __m128 b)
DIVSD	__m128d _mm_div_sd(__m128d a, __m128d b)
DIVSS	__m128 _mm_div_ss(__m128 a, __m128 b)
DPPD	__m128d _mm_dp_pd(__m128d a, __m128d b, const int mask)
DPPS	__m128 _mm_dp_ps(__m128 a, __m128 b, const int mask)
EMMS	void _mm_empty()
EXTRACTPS	int _mm_extract_ps(__m128 src, const int ndx)
HADDPD	__m128d _mm_hadd_pd(__m128d a, __m128d b)
HADDPS	__m128 _mm_hadd_ps(__m128 a, __m128 b)
HSUBPD	__m128d _mm_hsub_pd(__m128d a, __m128d b)
HSUBPS	__m128 _mm_hsub_ps(__m128 a, __m128 b)
INSERTPS	__m128 _mm_insert_ps(__m128 dst, __m128 src, const int ndx)
LDDQU	__m128i _mm_lddqu_si128(__m128i const *p)
LDMXCSR	__mm_setcsr(unsigned int i)
LFENCE	void _mm_lfence(void)
MASKMOVDQU	void _mm_maskmoveu_si128(__m128i d, __m128i n, char *p)
MASKMOVQ	void _mm_maskmove_si64(__m64 d, __m64 n, char *p)
MAXPD	__m128d _mm_max_pd(__m128d a, __m128d b)
MAXPS	__m128 _mm_max_ps(__m128 a, __m128 b)
MAXSD	__m128d _mm_max_sd(__m128d a, __m128d b)
MAXSS	__m128 _mm_max_ss(__m128 a, __m128 b)
MFENCE	void _mm_mfence(void)
MINPD	__m128d _mm_min_pd(__m128d a, __m128d b)
MINPS	__m128 _mm_min_ps(__m128 a, __m128 b)
MINSD	__m128d _mm_min_sd(__m128d a, __m128d b)
MINSS	__m128 _mm_min_ss(__m128 a, __m128 b)
MONITOR	void _mm_monitor(void const *p, unsigned extensions, unsigned hints)
MOVAPD	__m128d _mm_load_pd(double * p)
	void_mm_store_pd(double *p, __m128d a)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
MOVAPS	__m128 _mm_load_ps(float * p)
	void_mm_store_ps(float *p, __m128 a)
MOVD	__m128i _mm_cvtsi32_si128(int a)
	int _mm_cvtsi128_si32(__m128i a)
	__m64 _mm_cvtsi32_si64(int a)
	int _mm_cvtsi64_si32(__m64 a)
MOVDDUP	__m128d _mm_movedup_pd(__m128d a)
	__m128d _mm_loaddup_pd(double const * dp)
MOVDQA	__m128i _mm_load_si128(__m128i * p)
	void_mm_store_si128(__m128i *p, __m128i a)
MOVDQU	__m128i _mm_loadu_si128(__m128i * p)
	void_mm_storeu_si128(__m128i *p, __m128i a)
MOVDQ2Q	__m64 _mm_movepi64_pi64(__m128i a)
MOVHLPS	__m128 _mm_movehl_ps(__m128 a, __m128 b)
MOVHPD	__m128d _mm_loadh_pd(__m128d a, double * p)
	void _mm_storeh_pd(double * p, __m128d a)
MOVHPS	__m128 _mm_loadh_pi(__m128 a, __m64 * p)
	void _mm_storeh_pi(__m64 * p, __m128 a)
MOVLPD	__m128d _mm_loadl_pd(__m128d a, double * p)
	void _mm_storel_pd(double * p, __m128d a)
MOVLPS	__m128 _mm_loadl_pi(__m128 a, __m64 *p)
	void_mm_storel_pi(__m64 * p, __m128 a)
MOVLHPS	__m128 _mm_movelh_ps(__m128 a, __m128 b)
MOVMSKPD	int _mm_movemask_pd(__m128d a)
MOVMSKPS	int _mm_movemask_ps(__m128 a)
MOVNTDQA	__m128i _mm_stream_load_si128(__m128i *p)
MOVNTDQ	void_mm_stream_si128(__m128i * p, __m128i a)
MOVNTPD	void_mm_stream_pd(double * p, __m128d a)
MOVNTPS	void_mm_stream_ps(float * p, __m128 a)
MOVNTI	void_mm_stream_si32(int * p, int a)
MOVNTQ	void_mm_stream_pi(__m64 * p, __m64 a)
MOVQ	__m128i _mm_loadl_epi64(__m128i * p)
	void_mm_storel_epi64(_m128i * p, __m128i a)
	__m128i _mm_move_epi64(__m128i a)
MOVQ2DQ	__m128i _mm_movpi64_epi64(__m64 a)
MOVSD	__m128d _mm_load_sd(double * p)
	void_mm_store_sd(double * p, __m128d a)
	__m128d _mm_move_sd(__m128d a, __m128d b)
MOVSHDUP	__m128 _mm_movehdup_ps(__m128 a)
MOVSLDUP	__m128 _mm_moveldup_ps(__m128 a)
MOVSS	__m128 _mm_load_ss(float * p)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
	void_mm_store_ss(float * p, __m128 a)
	__m128 _mm_move_ss(__m128 a, __m128 b)
MOVUPD	__m128d _mm_loadu_pd(double * p)
	void_mm_storeu_pd(double *p, __m128d a)
MOVUPS	__m128 _mm_loadu_ps(float * p)
	void_mm_storeu_ps(float *p, __m128 a)
MPSADBW	__m128i _mm_mpsadbw_epu8(__m128i s1, __m128i s2, const int mask)
MULPD	__m128d _mm_mul_pd(__m128d a, __m128d b)
MULPS	__m128 _mm_mul_ss(__m128 a, __m128 b)
MULSD	__m128d _mm_mul_sd(__m128d a, __m128d b)
MULSS	__m128 _mm_mul_ss(__m128 a, __m128 b)
MWAIT	void _mm_mwait(unsigned extensions, unsigned hints)
ORPD	__m128d _mm_or_pd(__m128d a, __m128d b)
ORPS	__m128 _mm_or_ps(__m128 a, __m128 b)
PABSB	__m64 _mm_abs_pi8 (__m64 a)
	__m128i _mm_abs_epi8 (__m128i a)
PABSD	__m64 _mm_abs_pi32 (__m64 a)
	__m128i _mm_abs_epi32 (__m128i a)
PABSW	__m64 _mm_abs_pi16 (__m64 a)
	__m128i _mm_abs_epi16 (__m128i a)
PACKSSWB	__m128i _mm_packs_epi16(__m128i m1, __m128i m2)
PACKSSWB	__m64 _mm_packs_pi16(__m64 m1, __m64 m2)
PACKSSDW	__m128i _mm_packs_epi32 (__m128i m1, __m128i m2)
PACKSSDW	__m64 _mm_packs_pi32 (__m64 m1, __m64 m2)
PACKUSDW	__m128i _mm_packus_epi32(__m128i m1, __m128i m2)
PACKUSWB	__m128i _mm_packus_epi16(__m128i m1, __m128i m2)
PACKUSWB	__m64 _mm_packs_pu16(__m64 m1, __m64 m2)
PADDB	__m128i _mm_add_epi8(__m128i m1, __m128i m2)
PADDB	__m64 _mm_add_pi8(__m64 m1, __m64 m2)
PADDW	__m128i _mm_add_epi16(__m128i m1, __m128i m2)
PADDW	__m64 _mm_add_pi16(__m64 m1, __m64 m2)
PADDD	__m128i _mm_add_epi32(__m128i m1, __m128i m2)
PADDD	__m64 _mm_add_pi32(__m64 m1, __m64 m2)
PADDQ	__m128i _mm_add_epi64(__m128i m1, __m128i m2)
PADDQ	__m64 _mm_add_si64(__m64 m1, __m64 m2)
PADDSB	__m128i _mm_adds_epi8(__m128i m1, __m128i m2)
PADDSB	__m64 _mm_adds_pi8(__m64 m1, __m64 m2)
PADDSW	__m128i _mm_adds_epi16(__m128i m1, __m128i m2)
PADDSW	__m64 _mm_adds_pi16(__m64 m1, __m64 m2)
PADDUSB	__m128i _mm_adds_epu8(__m128i m1, __m128i m2)
PADDUSB	__m64 _mm_adds_pu8(__m64 m1, __m64 m2)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
PADDUSW	__m128i _mm_adds_epu16(__m128i m1, __m128i m2)
PADDUSW	__m64 _mm_adds_pu16(__m64 m1, __m64 m2)
PALIGNR	__m64 _mm_alignr_pi8 (__m64 a, __m64 b, int n)
	__m128i _mm_alignr_epi8 (__m128i a, __m128i b, int n)
PAND	__m128i _mm_and_si128(__m128i m1, __m128i m2)
PAND	__m64 _mm_and_si64(__m64 m1, __m64 m2)
PANDN	__m128i _mm_andnot_si128(__m128i m1, __m128i m2)
PANDN	__m64 _mm_andnot_si64(__m64 m1, __m64 m2)
PAUSE	void _mm_pause(void)
PAVGB	__m128i _mm_avg_epu8(__m128i a, __m128i b)
PAVGB	__m64 _mm_avg_pu8(__m64 a, __m64 b)
PAVGW	__m128i _mm_avg_epu16(__m128i a, __m128i b)
PAVGW	__m64 _mm_avg_pu16(__m64 a, __m64 b)
PBLENDVB	__m128i _mm_blendv_epi (__m128i v1, __m128i v2, __m128i mask)
PBLENDW	__m128i _mm_blend_epi16(__m128i v1, __m128i v2, const int mask)
PCLMULQDQ	__m128i _mm_clmulepi64_si128 (__m128i, __m128i, const int)
PCMPEQB	__m128i _mm_cmpeq_epi8(__m128i m1, __m128i m2)
PCMPEQB	__m64 _mm_cmpeq_pi8(__m64 m1, __m64 m2)
PCMPEQQ	__m128i _mm_cmpeq_epi64(__m128i a, __m128i b)
PCMPEQW	__m128i _mm_cmpeq_epi16 (__m128i m1, __m128i m2)
PCMPEQW	__m64 _mm_cmpeq_pi16 (__m64 m1, __m64 m2)
PCMPEQD	__m128i _mm_cmpeq_epi32(__m128i m1, __m128i m2)
PCMPEQD	__m64 _mm_cmpeq_pi32(__m64 m1, __m64 m2)
PCMPESTRI	int _mm_cmpestri (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestra (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestrc (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestrz (__m128i a, int la, __m128i b, int lb, const int mode)
PCMPESTRM	__m128i _mm_cmpestrm (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestra (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestrc (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestrz (__m128i a, int la, __m128i b, int lb, const int mode)
PCMPGTB	__m128i _mm_cmpgt_epi8 (__m128i m1, __m128i m2)
PCMPGTB	__m64 _mm_cmpgt_pi8 (__m64 m1, __m64 m2)
PCMPGTW	__m128i _mm_cmpgt_epi16(__m128i m1, __m128i m2)
PCMPGTW	__m64 _mm_cmpgt_pi16 (__m64 m1, __m64 m2)
PCMPGTD	__m128i _mm_cmpgt_epi32(__m128i m1, __m128i m2)
PCMPGTD	__m64 _mm_cmpgt_pi32(__m64 m1, __m64 m2)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
PCMPISTRI	__m128i _mm_cmpestrm (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestra (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestrc (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpistrz (__m128i a, __m128i b, const int mode)
PCMPISTRM	__m128i _mm_cmpistrm (__m128i a, __m128i b, const int mode)
	int _mm_cmpistra (__m128i a, __m128i b, const int mode)
	int _mm_cmpistrc (__m128i a, __m128i b, const int mode)
	int _mm_cmpistro (__m128i a, __m128i b, const int mode)
	int _mm_cmpistrs (__m128i a, __m128i b, const int mode)
	int _mm_cmpistrz (__m128i a, __m128i b, const int mode)
PCMPGTQ	__m128i _mm_cmpgt_epi64(__m128i a, __m128i b)
PEXTRB	int _mm_extract_epi8 (__m128i src, const int ndx)
PEXTRD	int _mm_extract_epi32 (__m128i src, const int ndx)
PEXTRQ	__int64 _mm_extract_epi64 (__m128i src, const int ndx)
PEXTRW	int _mm_extract_epi16(__m128i a, int n)
PEXTRW	int _mm_extract_pi16(__m64 a, int n)
	int _mm_extract_epi16 (__m128i src, int ndx)
PHADDD	__m64 _mm_hadd_pi32 (__m64 a, __m64 b)
	__m128i _mm_hadd_epi32 (__m128i a, __m128i b)
PHADDSW	__m64 _mm_hadds_pi16 (__m64 a, __m64 b)
	__m128i _mm_hadds_epi16 (__m128i a, __m128i b)
PHADDW	__m64 _mm_hadd_pi16 (__m64 a, __m64 b)
	__m128i _mm_hadd_epi16 (__m128i a, __m128i b)
PHMINPOSUW	__m128i _mm_minpos_epu16(__m128i packed_words)
PHSUBD	__m64 _mm_hsub_pi32 (__m64 a, __m64 b)
	__m128i _mm_hsub_epi32 (__m128i a, __m128i b)
PHSUBSW	__m64 _mm_hsubs_pi16 (__m64 a, __m64 b)
	__m128i _mm_hsubs_epi16 (__m128i a, __m128i b)
PHSUBW	__m64 _mm_hsub_pi16 (__m64 a, __m64 b)
	__m128i _mm_hsub_epi16 (__m128i a, __m128i b)
PINSRB	__m128i _mm_insert_epi8(__m128i s1, int s2, const int ndx)
PINSRD	__m128i _mm_insert_epi32(__m128i s2, int s, const int ndx)
PINSRQ	__m128i _mm_insert_epi64(__m128i s2, __int64 s, const int ndx)
PINSRW	__m128i _mm_insert_epi16(__m128i a, int d, int n)
PINSRW	__m64 _mm_insert_pi16(__m64 a, int d, int n)
PMADDUBSW	__m64 _mm_maddubs_pi16 (__m64 a, __m64 b)
	__m128i _mm_maddubs_epi16 (__m128i a, __m128i b)
PMADDWD	__m128i _mm_madd_epi16(__m128i m1 __m128i m2)
PMADDWD	__m64 _mm_madd_pi16(__m64 m1, __m64 m2)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
PMAXSB	__m128i _mm_max_epi8(__m128i a, __m128i b)
PMAXSD	__m128i _mm_max_epi32(__m128i a, __m128i b)
PMAXSW	__m128i _mm_max_epi16(__m128i a, __m128i b)
PMAXSW	__m64 _mm_max_pi16(__m64 a, __m64 b)
PMAXUB	__m128i _mm_max_epu8(__m128i a, __m128i b)
PMAXUB	__m64 _mm_max_pu8(__m64 a, __m64 b)
PMAXUD	__m128i _mm_max_epu32(__m128i a, __m128i b)
PMAXUW	__m128i _mm_max_epu16(__m128i a, __m128i b)
PMINSB	_m128i _mm_min_epi8(__m128i a, __m128i b)
PMINSD	__m128i _mm_min_epi32(__m128i a, __m128i b)
PMINSW	__m128i _mm_min_epi16(__m128i a, __m128i b)
PMINSW	__m64 _mm_min_pi16(__m64 a, __m64 b)
PMINUB	__m128i _mm_min_epu8(__m128i a, __m128i b)
PMINUB	__m64 _mm_min_pu8(__m64 a, __m64 b)
PMINUD	__m128i _mm_min_epu32 (__m128i a, __m128i b)
PMINUW	__m128i _mm_min_epu16 (__m128i a, __m128i b)
PMOVMSKB	int _mm_movemask_epi8(__m128i a)
PMOVMSKB	int _mm_movemask_pi8(__m64 a)
PMOVSXBW	__m128i _mm_ cvtepi8_epi16(__m128i a)
PMOVSXBD	__m128i _mm_ cvtepi8_epi32(__m128i a)
PMOVSXBQ	__m128i _mm_ cvtepi8_epi64(__m128i a)
PMOVSXWD	__m128i _mm_ cvtepi16_epi32(__m128i a)
PMOVSXWQ	__m128i _mm_ cvtepi16_epi64(__m128i a)
PMOVSXDQ	__m128i _mm_ cvtepi32_epi64(__m128i a)
PMOVZXBW	__m128i _mm_ cvtepu8_epi16(__m128i a)
PMOVZXBD	__m128i _mm_ cvtepu8_epi32(__m128i a)
PMOVZXBQ	__m128i _mm_ cvtepu8_epi64(__m128i a)
PMOVZXWD	__m128i _mm_ cvtepu16_epi32(__m128i a)
PMOVZXWQ	__m128i _mm_ cvtepu16_epi64(__m128i a)
PMOVZXDQ	__m128i _mm_ cvtepu32_epi64(__m128i a)
PMULDQ	__m128i _mm_mul_epi32(__m128i a, __m128i b)
PMULHRSW	__m64 _mm_mulhrs_pi16 (__m64 a, __m64 b)
	__m128i _mm_mulhrs_epi16 (__m128i a, __m128i b)
PMULHUW	__m128i _mm_mulhi_epu16(__m128i a, __m128i b)
PMULHUW	__m64 _mm_mulhi_pu16(__m64 a, __m64 b)
PMULHW	__m128i _mm_mulhi_epi16(__m128i m1, __m128i m2)
PMULHW	__m64 _mm_mulhi_pi16(__m64 m1, __m64 m2)
PMULLUD	__m128i _mm_mullo_epi32(__m128i a, __m128i b)
PMULLW	__m128i _mm_mullo_epi16(__m128i m1, __m128i m2)
PMULLW	__m64 _mm_mullo_pi16(__m64 m1, __m64 m2)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
PMULUDQ	__m64 _mm_mul_su32(__m64 m1, __m64 m2)
	__m128i _mm_mul_epu32(__m128i m1, __m128i m2)
POPCNT	int _mm_popcnt_u32(unsigned int a)
	int64_t _mm_popcnt_u64(unsigned __int64 a)
POR	__m64 _mm_or_si64(__m64 m1, __m64 m2)
POR	__m128i _mm_or_si128(__m128i m1, __m128i m2)
PREFETCHh	void _mm_prefetch(char *a, int sel)
PSADBW	__m128i _mm_sad_epu8(__m128i a, __m128i b)
PSADBW	__m64 _mm_sad_pu8(__m64 a, __m64 b)
PSHUFB	__m64 _mm_shuffle_pi8 (__m64 a, __m64 b)
	__m128i _mm_shuffle_epi8 (__m128i a, __m128i b)
PSHUFD	__m128i _mm_shuffle_epi32(__m128i a, int n)
PSHUFHW	__m128i _mm_shufflehi_epi16(__m128i a, int n)
PSHUFLW	__m128i _mm_shufflelo_epi16(__m128i a, int n)
PSHUFW	__m64 _mm_shuffle_pi16(__m64 a, int n)
PSIGNB	__m64 _mm_sign_pi8 (__m64 a, __m64 b)
	__m128i _mm_sign_epi8 (__m128i a, __m128i b)
PSIGND	__m64 _mm_sign_pi32 (__m64 a, __m64 b)
	__m128i _mm_sign_epi32 (__m128i a, __m128i b)
PSIGNW	__m64 _mm_sign_pi16 (__m64 a, __m64 b)
	__m128i _mm_sign_epi16 (__m128i a, __m128i b)
PSLLW	__m128i _mm_sll_epi16(__m128i m, __m128i count)
PSLLW	__m128i _mm_slli_epi16(__m128i m, int count)
PSLLW	__m64 _mm_sll_pi16(__m64 m, __m64 count)
	__m64 _mm_slli_pi16(__m64 m, int count)
PSLLD	__m128i _mm_slli_epi32(__m128i m, int count)
	__m128i _mm_sll_epi32(__m128i m, __m128i count)
PSLLD	__m64 _mm_slli_pi32(__m64 m, int count)
	__m64 _mm_sll_pi32(__m64 m, __m64 count)
PSLLQ	__m64 _mm_sll_si64(__m64 m, __m64 count)
	__m64 _mm_slli_si64(__m64 m, int count)
PSLLQ	__m128i _mm_sll_epi64(__m128i m, __m128i count)
	__m128i _mm_slli_epi64(__m128i m, int count)
PSLLDQ	__m128i _mm_slli_si128(__m128i m, int imm)
PSRAW	__m128i _mm_sra_epi16(__m128i m, __m128i count)
	__m128i _mm_srai_epi16(__m128i m, int count)
PSRAW	__m64 _mm_sra_pi16(__m64 m, __m64 count)
	__m64 _mm_srai_pi16(__m64 m, int count)
PSRAD	__m128i _mm_sra_epi32 (__m128i m, __m128i count)
	__m128i _mm_srai_epi32 (__m128i m, int count)
PSRAD	__m64 _mm_sra_pi32 (__m64 m, __m64 count)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
	__m64 _mm_srai_pi32 (__m64 m, int count)
PSRLW	_m128i _mm_srl_epi16 (__m128i m, __m128i count)
	__m128i _mm_srli_epi16 (__m128i m, int count)
	__m64 _mm_srl_pi16 (__m64 m, __m64 count)
	__m64 _mm_srli_pi16(__m64 m, int count)
PSRLD	__m128i _mm_srl_epi32 (__m128i m, __m128i count)
	__m128i _mm_srli_epi32 (__m128i m, int count)
PSRLD	__m64 _mm_srl_pi32 (__m64 m, __m64 count)
	__m64 _mm_srli_pi32 (__m64 m, int count)
PSRLQ	__m128i _mm_srl_epi64 (__m128i m, __m128i count)
	__m128i _mm_srli_epi64 (__m128i m, int count)
PSRLQ	__m64 _mm_srl_si64 (__m64 m, __m64 count)
	__m64 _mm_srli_si64 (__m64 m, int count)
PSRLDQ	__m128i _mm_srli_si128(__m128i m, int imm)
PSUBB	__m128i _mm_sub_epi8(__m128i m1, __m128i m2)
PSUBB	__m64 _mm_sub_pi8(__m64 m1, __m64 m2)
PSUBW	__m128i _mm_sub_epi16(__m128i m1, __m128i m2)
PSUBW	__m64 _mm_sub_pi16(__m64 m1, __m64 m2)
PSUBD	__m128i _mm_sub_epi32(__m128i m1, __m128i m2)
PSUBD	__m64 _mm_sub_pi32(__m64 m1, __m64 m2)
PSUBQ	__m128i _mm_sub_epi64(__m128i m1, __m128i m2)
PSUBQ	__m64 _mm_sub_si64(__m64 m1, __m64 m2)
PSUBSB	__m128i _mm_subs_epi8(__m128i m1, __m128i m2)
PSUBSB	__m64 _mm_subs_pi8(__m64 m1, __m64 m2)
PSUBSW	__m128i _mm_subs_epi16(__m128i m1, __m128i m2)
PSUBSW	__m64 _mm_subs_pi16(__m64 m1, __m64 m2)
PSUBUSB	__m128i _mm_subs_epu8(__m128i m1, __m128i m2)
PSUBUSB	__m64 _mm_subs_pu8(__m64 m1, __m64 m2)
PSUBUSW	__m128i _mm_subs_epu16(__m128i m1, __m128i m2)
PSUBUSW	__m64 _mm_subs_pu16(__m64 m1, __m64 m2)
PTEST	int _mm_testz_si128(__m128i s1, __m128i s2)
	int _mm_testc_si128(__m128i s1, __m128i s2)
	int _mm_testnzc_si128(__m128i s1, __m128i s2)
PUNPCKHBW	__m64 _mm_unpackhi_pi8(__m64 m1, __m64 m2)
PUNPCKHBW	__m128i _mm_unpackhi_epi8(__m128i m1, __m128i m2)
PUNPCKHWD	__m64 _mm_unpackhi_pi16(__m64 m1,__m64 m2)
PUNPCKHWD	__m128i _mm_unpackhi_epi16(__m128i m1, __m128i m2)
PUNPCKHDQ	___m64 _mm_unpackhi_pi32(__m64 m1, __m64 m2)
PUNPCKHDQ	__m128i _mm_unpackhi_epi32(__m128i m1, __m128i m2)
PUNPCKHQDQ	__m128i _mm_unpackhi_epi64(__m128i m1, __m128i m2)
PUNPCKLBW	__m64 _mm_unpacklo_pi8 (__m64 m1, __m64 m2)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
PUNPCKLBW	__m128i _mm_unpacklo_epi8 (__m128i m1, __m128i m2)
PUNPCKLWD	__m64 _mm_unpacklo_pi16(__m64 m1, __m64 m2)
PUNPCKLWD	__m128i _mm_unpacklo_epi16(__m128i m1, __m128i m2)
PUNPCKLDQ	__m64 _mm_unpacklo_pi32(__m64 m1, __m64 m2)
PUNPCKLDQ	__m128i _mm_unpacklo_epi32(__m128i m1, __m128i m2)
PUNPCKLQDQ	__m128i _mm_unpacklo_epi64(__m128i m1, __m128i m2)
PXOR	__m64 _mm_xor_si64(__m64 m1, __m64 m2)
PXOR	__m128i _mm_xor_si128(__m128i m1, __m128i m2)
RCPPS	__m128 _mm_rcp_ps(__m128 a)
RCPSS	__m128 _mm_rcp_ss(__m128 a)
ROUNDPD	__m128 mm_round_pd(__m128d s1, int iRoundMode)
	__m128 mm_floor_pd(__m128d s1)
	__m128 mm_ceil_pd(__m128d s1)
ROUNDPS	__m128 mm_round_ps(__m128 s1, int iRoundMode)
	__m128 mm_floor_ps(__m128 s1)
	__m128 mm_ceil_ps(__m128 s1)
ROUNDSD	__m128d mm_round_sd(__m128d dst, __m128d s1, int iRoundMode)
	__m128d mm_floor_sd(__m128d dst, __m128d s1)
	__m128d mm_ceil_sd(__m128d dst, __m128d s1)
ROUNDSS	__m128 mm_round_ss(__m128 dst, __m128 s1, int iRoundMode)
	__m128 mm_floor_ss(__m128 dst, __m128 s1)
	__m128 mm_ceil_ss(__m128 dst, __m128 s1)
RSQRTPS	__m128 _mm_rsqrt_ps(__m128 a)
RSQRTSS	__m128 _mm_rsqrt_ss(__m128 a)
SFENCE	void_mm_sfence(void)
SHUFPD	__m128d _mm_shuffle_pd(__m128d a, __m128d b, unsigned int imm8)
SHUFPS	__m128 _mm_shuffle_ps(__m128 a, __m128 b, unsigned int imm8)
SQRTPD	__m128d _mm_sqrt_pd(__m128d a)
SQRTPS	__m128 _mm_sqrt_ps(__m128 a)
SQRTSD	__m128d _mm_sqrt_sd(__m128d a)
SQRTSS	__m128 _mm_sqrt_ss(__m128 a)
STMXCSR	_mm_getcsr(void)
SUBPD	__m128d _mm_sub_pd(__m128d a, __m128d b)
SUBPS	__m128 _mm_sub_ps(__m128 a, __m128 b)
SUBSD	__m128d _mm_sub_sd(__m128d a, __m128d b)
SUBSS	__m128 _mm_sub_ss(__m128 a, __m128 b)
UCOMISD	int _mm_ucomieq_sd(__m128d a, __m128d b)
	int _mm_ucomilt_sd(__m128d a, __m128d b)
	int _mm_ucomile_sd(__m128d a, __m128d b)
	int _mm_ucomigt_sd(__m128d a, __m128d b)
	int _mm_ucomige_sd(__m128d a, __m128d b)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
	int _mm_ucomineq_sd(__m128d a, __m128d b)
UCOMISS	int _mm_ucomieq_ss(__m128 a, __m128 b)
	int _mm_ucomilt_ss(__m128 a, __m128 b)
	int _mm_ucomile_ss(__m128 a, __m128 b)
	int _mm_ucomigt_ss(__m128 a, __m128 b)
	int _mm_ucomige_ss(__m128 a, __m128 b)
	int _mm_ucomineq_ss(__m128 a, __m128 b)
UNPCKHPD	__m128d _mm_unpackhi_pd(__m128d a, __m128d b)
UNPCKHPS	__m128 _mm_unpackhi_ps(__m128 a, __m128 b)
UNPCKLPD	__m128d _mm_unpacklo_pd(__m128d a, __m128d b)
UNPCKLPS	__m128 _mm_unpacklo_ps(__m128 a, __m128 b)
XORPD	__m128d _mm_xor_pd(__m128d a, __m128d b)
XORPS	__m128 _mm_xor_ps(__m128 a, __m128 b)

C.2 COMPOSITE INTRINSICS

Table C-2. Composite Intrinsics

Mnemonic	Intrinsic
(composite)	__m128i _mm_set_epi64(__m64 q1, __m64 q0)
(composite)	__m128i _mm_set_epi32(int i3, int i2, int i1, int i0)
(composite)	__m128i _mm_set_epi16(short w7,short w6, short w5, short w4, short w3, short w2, short w1,short w0)
(composite)	__m128i _mm_set_epi8(char w15,char w14, char w13, char w12, char w11, char w10, char w9, char w8, char w7,char w6, char w5, char w4, char w3, char w2,char w1, char w0)
(composite)	__m128i _mm_set1_epi64(__m64 q)
(composite)	__m128i _mm_set1_epi32(int a)
(composite)	__m128i _mm_set1_epi16(short a)
(composite)	__m128i _mm_set1_epi8(char a)
(composite)	__m128i _mm_setr_epi64(__m64 q1, __m64 q0)
(composite)	__m128i _mm_setr_epi32(int i3, int i2, int i1, int i0)
(composite)	__m128i _mm_setr_epi16(short w7,short w6, short w5, short w4, short w3, short w2, short w, short w0)
(composite)	__m128i _mm_setr_epi8(char w15,char w14, char w13, char w12, char w11, char w10, char w9, char w8,char w7, char w6,char w5, char w4, char w3, char w2,char w1,char w0)
(composite)	__m128i _mm_setzero_si128()
(composite)	__m128 _mm_set_ps1(float w) __m128 _mm_set1_ps(float w)
(composite)	__m128cmm_set1_pd(double w)
(composite)	__m128d _mm_set_sd(double w)
(composite)	__m128d _mm_set_pd(double z, double y)
(composite)	__m128 _mm_set_ps(float z, float y, float x, float w)
(composite)	__m128d _mm_setr_pd(double z, double y)
(composite)	__m128 _mm_setr_ps(float z, float y, float x, float w)

Table C-2. Composite Intrinsics (Contd.)

Mnemonic	Intrinsic
(composite)	__m128d _mm_setzero_pd(void)
(composite)	__m128 _mm_setzero_ps(void)
MOVSD + shuffle	__m128d _mm_load_pd(double * p) __m128d _mm_load1_pd(double *p)
MOVSS + shuffle	__m128 _mm_load_ps1(float * p) __m128 _mm_load1_ps(float *p)
MOVAPD + shuffle	__m128d _mm_loadr_pd(double * p)
MOVAPS + shuffle	__m128 _mm_loadr_ps(float * p)
MOVSD + shuffle	void _mm_store1_pd(double *p, __m128d a)
MOVSS + shuffle	void _mm_store_ps1(float * p, __m128 a) void _mm_store1_ps(float *p, __m128 a)
MOVAPD + shuffle	_mm_storer_pd(double * p, __m128d a)
MOVAPS + shuffle	_mm_storer_ps(float * p, __m128 a)

Numerics

0000, B-41
MADD, 5-214, 5-229
MSUB, 5-263, 5-279
64-bit mode
  control and debug registers, 2-12
  default operand size, 2-12
  direct memory-offset MOVs, 2-11
  general purpose encodings, B-18
  immediates, 2-11
  introduction, 2-7
  machine instructions, B-1
  reg (reg) field, B-4
  REX prefixes, 2-7, B-2
  RIP-relative addressing, 2-11
  SIMD encodings, B-37
  special instruction encodings, B-61
  summary table notation, 3-8

A

AAA instruction, 3-1, 3-3
AAD instruction, 3-3
AAM instruction, 3-5
AAS instruction, 3-7
ADC instruction, 3-9, 3-565
ADD instruction, 3-1, 3-14, 3-258, 3-565
ADDPD instruction, 3-16
ADDPS- Add Packed Single Precision Floating-Point Values, 3-19
Addressing methods
  RIP-relative, 2-11
ADDSD- Add Scalar Double Precision Floating-Point Values, 3-22
ADDSS- Add Scalar Single Precision Floating-Point Values, 3-24
ADDSUBPD instruction, 3-26
ADDSUBPS instruction, 3-28
ADOX — Unsigned Integer Addition of Two Operands with Overflow Flag, 3-31
AESDEC128KL—Perform Ten Rounds of AES Decryption Flow Using 128-Bit Key, 3-35
AESDEC256KL—Perform 14 Rounds of AES Decryption Flow Using 256-Bit Key, 3-37
AESDECLAST—Perform Last Round of an AES Decryption Flow, 3-39
AESDEC—Perform One Round of an AES Decryption Flow, 3-33, 4-763
AESDECWIDE128KL—Perform Ten Rounds of AES Decryption Flow on 8 Blocks with 128-Bit Key, 3-41
AESDECWIDE256KL—Perform 14 Rounds of AES Decryption Flow on 8 Blocks with 256-Bit Key, 3-43
AESENC128KL—Perform Ten Rounds of AES Encryption Flow Using 128-Bit Key, 3-47
AESENC256KL—Perform 14 Rounds of AES Encryption Flow Using 256-Bit Key, 3-49
AESENCLAST—Perform Last Round of an AES Encryption Flow, 3-51
AESENC—Perform One Round of an AES Encryption Flow, 3-45
AESENCWIDE128KL—Perform Ten Rounds of AES Encryption Flow on 8 Blocks with 128-Bit Key, 3-53
AESENCWIDE256KL—Perform 14 Rounds of AES Encryption Flow on 8 Blocks with 256-Bit Key, 3-55
AESIMC—Perform the AES InvMixColumn Transformation, 3-57
AESKEYGENASSIST - AES Round Key Generation Assist, 3-58
AND instruction, 3-60, 3-565
ANDNPS- Bitwise Logical AND NOT of Packed Single Precision Floating-Point Values, 3-66
ANDPD- Bitwise Logical AND of Packed Double Precision Floating-Point Values, 3-69
ANDPD instruction, 3-62
ANDPS- Bitwise Logical AND of Packed Single Precision Floating-Point Values, 3-72
Arctangent, x87 FPU operation, 3-364
ARPL instruction, 3-75
authenticated code execution mode, 7-3

B

Base (operand addressing), 2-3
BCD integers
  packed, 3-258, 3-260, 3-314, 3-316
  unpacked, 3-1, 3-3, 3-5, 3-7
BEXTR—Bit Field Extract, 3-77
BLENDPD — Blend Packed Double Precision Floating-Point Values, 3-78
BLENDPS — Blend Packed Single Precision Floating-Point Values, 3-80
BLSI-Extract Lowest Set Isolated Bit, 3-87
BLSMSK - Get Mask Up to Lowest Set Bit, 3-88
BLSR —Reset Lowest Set Bit, 3-89
BNDCL—Check Lower Bound, 3-90
BNDCU/BNDCN—Check Upper Bound, 3-92
BNDLDX—Load Extended Bounds Using Address Translation, 3-94
BNDMK—Make Bounds, 3-97
BNDMOV—Move Bounds, 3-99
BNDSTX—Store Extended Bounds Using Address Translation, 3-102
bootstrap processor, 7-16, 7-22, 7-30, 7-31
BOUND instruction, 3-105
BOUND range exceeded exception (#BR), 3-105
BOUND—Check Array Index Against Bounds, 3-105
Branch hints, 2-2
BSF instruction, 3-107
BSR instruction, 3-109
BSWAP instruction, 3-111
BT instruction, 3-112
BTC instruction, 3-114, 3-565
BTR instruction, 3-116, 3-565
BTS instruction, 3-118, 3-565
BZHI —Zero High Bits Starting with Specified Bit Position, 3-120

C

Caches, invalidating (flushing), 3-483, 6-2
CALL instruction, 3-121
GETSEC, 7-3
CBW instruction, 3-138
CDQ instruction, 3-257
CDQE instruction, 3-138
CF (carry) flag, EFLAGS register, 3-14, 3-112, 3-114, 3-116, 3-118, 3-140, 3-156, 3-262, 3-452, 3-457, 4-140, 4-534, 4-613, 4-640, 4-643, 4-685
CLC instruction, 3-140
CLD instruction, 3-141
CLFLUSH instruction, 3-144, 3-146
CLI instruction, 3-148
CLTS instruction, 3-152
CMC instruction, 3-156
CMOVcc instructions, 3-157
CMP instruction, 3-161
CMPPD- Compare Packed Double Precision Floating-Point Values, 3-168

CMPPS- Compare Packed Single Precision Floating-Point Values, 3-175
CMPS instruction, 3-181, 4-567
CMPSB instruction, 3-181
CMPSD- Compare Scalar Double Precision Floating-Point Values, 3-185
CMPSD instruction, 3-181
CMPSQ instruction, 3-181
CMPSS- Compare Scalar Single Precision Floating-Point Values, 3-189
CMPSW instruction, 3-181
CMPXCHG instruction, 3-194, 3-565
CMPXCHG16B instruction, 3-196
CMPXCHG8B instruction, 3-196
COMISD- Compare Scalar Ordered Double Precision Floating-Point Values and Set EFLAGS, 3-199
COMISS- Compare Scalar Ordered Single Precision Floating-Point Values and Set EFLAGS, 3-201
Compatibility mode
  introduction, 2-7
  see 64-bit mode
  summary table notation, 3-9
Condition code flags, EFLAGS register, 3-157
Condition code flags, x87 FPU status word
  flags affected by instructions, 3-15
  setting, 3-400, 3-402, 3-405
Conditional jump, 3-499
Conforming code segment, 3-534
Constants (floating point), loading, 3-354
Control registers, moving values to and from, 4-32
Cosine, x87 FPU operation, 3-330, 3-382
CPL, 3-148, 5-162
CPUID instruction, 3-203
  using CPUID, 3-203
CQO instruction, 3-257
CR0 control register, 4-659
CS register, 3-122, 3-468, 3-490, 3-505, 4-29, 4-399
CVTDQ2PD- Convert Packed Doubleword Integers to Packed Double Precision Floating-Point Values, 3-208
CVTDQ2PD instruction, 3-205
CVTDQ2PS- Convert Packed Doubleword Integers to Packed Single Precision Floating-Point Values, 3-211
CVTPD2DQ- Convert Packed Double Precision Floating-Point Values to Packed Doubleword Integers, 3-214
CVTPD2PI instruction, 3-218
CVTPD2PS- Convert Packed Double Precision Floating-Point Values to Packed Single Precision Floating-Point Values, 3-219
CVTPI2PD instruction, 3-223
CVTPI2PS instruction, 3-224
CVTPS2DQ- Convert Packed Single Precision Floating-Point Values to Packed Signed Doubleword Integer Values, 3-225
CVTPS2PI instruction, 3-231
CVTSD2SI- Convert Scalar Double Precision Floating-Point Value to Doubleword Integer, 3-232
CVTSI2SD- Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value, 5-138
CVTSI2SS- Convert Doubleword Integer to Scalar Single Precision Floating-Point Value, 3-238
CVTSS2SD- Convert Scalar Single Precision Floating-Point Value to Scalar Double Precision Floating-Point Value, 3-240
CVTSS2SI- Convert Scalar Single Precision Floating-Point Value to Doubleword Integer, 3-242
CVTTPD2DQ- Convert with Truncation Packed Double Precision Floating-Point Values to Packed Doubleword Integers, 3-244
CVTTPD2PI instruction, 3-248
CVTTPS2DQ- Convert with Truncation Packed Single Precision Floating-Point Values to Packed Signed Doubleword Integer Values, 3-249
CVTTPS2PI instruction, 3-252
CVTTSD2SI- Convert with Truncation Scalar Double Precision Floating-Point Value to Signed Integer, 3-253
CVTTSS2SI- Convert with Truncation Scalar Single Precision Floating-Point Value to Integer, 3-255
CWD instruction, 3-257
CWDE instruction, 3-138
C/C++ compiler intrinsics
  compiler functional equivalents, C-1
  composite, C-14
  description of, 3-12
  lists of, C-1
  simple, C-2

D

D (default operation size) flag, segment descriptor, 4-403
DAA instruction, 3-258
DAS instruction, 3-260
Debug registers, moving value to and from, 4-35
DEC instruction, 3-262, 3-565
Denormalized finite number, 3-405
Detecting and Enabling SMX
  level 2, 7-1
DF (direction) flag, EFLAGS register, 3-141, 3-182, 3-464, 3-567, 4-104, 4-173, 4-616, 4-673
Displacement (operand addressing), 2-3
DIV instruction, 3-264
Divide error exception (#DE), 3-264
DIVPD- Divide Packed Double Precision Floating-Point Values, 3-267
DIVPS- Divide Packed Single Precision Floating-Point Values, 3-270
DIVSD- Divide Scalar Double Precision Floating-Point Values, 3-273
DIVSS- Divide Scalar Single Precision Floating-Point Values, 3-275
DS register, 3-181, 3-540, 3-567, 4-104, 4-173

E

EDI register, 4-616, 4-673, 4-677
Effective address, 3-547
EFLAGS register
  condition codes, 3-159, 3-322, 3-327
  flags affected by instructions, 3-14
  popping, 4-407
  popping on return from interrupt, 3-490
  pushing, 4-528
  pushing on interrupts, 3-468
  saving, 4-601
  status flags, 3-161, 3-502, 4-624, 4-721
EIP register, 3-122, 3-468, 3-490, 3-505
EMMS instruction, 3-282
ENCODEKEY128—Encode 128-Bit Key, 3-283
ENCODEKEY256—Encode 256-Bit Key, 3-285
Encodings
  See machine instructions, opcodes
ENDBR32—Terminate an Indirect Branch in 32-bit and Compatibility Mode, 3-287
ENTER instruction, 3-295

GETSEC, 7-3, 7-10
ES register, 3-540, 4-173, 4-616, 4-677
ESI register, 3-181, 3-567, 4-104, 4-173, 4-673
ESP register, 3-122
EVEX.R, 3-5
Exceptions
  BOUND range exceeded (#BR), 3-105
  overflow exception (#OF), 3-467
  returning from, 3-298, 3-301, 3-490, 3-556
GETSEC, 7-3, 7-5
Exponent, extracting from floating-point number, 3-420
Extract exponent and significand, x87 FPU operation, 3-420
EXTRACTPS- Extract packed floating-point values, 3-305

F

F2XM1 instruction, 3-307, 3-420
FABS instruction, 3-309
FADD instruction, 3-311
FADDP instruction, 3-311
Far pointer, loading, 3-540
Far return, RET instruction, 4-569
FBLD instruction, 3-314
FBSTP instruction, 3-316
FCHS instruction, 3-318
FCLEX instruction, 3-320
FCMOVcc instructions, 3-322
FCOM instruction, 3-324
FCOMI instruction, 3-327
FCOMIP instruction, 3-327
FCOMP instruction, 3-324
FCOMPP instruction, 3-324
FCOS instruction, 3-330
FDECSTP instruction, 3-332
FDIV instruction, 3-333
FDIVP instruction, 3-333
FDIVR instruction, 3-336
FDIVRP instruction, 3-336
Feature information, processor, 3-203
FFREE instruction, 3-339
FIADD instruction, 3-311
FICOM instruction, 3-340
FICOMP instruction, 3-340
FIDIV instruction, 3-333
FIDIVR instruction, 3-336
FILD instruction, 3-342
FIMUL instruction, 3-360
FINCSTP instruction, 3-344
FINIT instruction, 3-345
FINIT/FNINIT instructions, 3-375
FIST instruction, 3-347
FISTP instruction, 3-347
FISTTP instruction, 3-350
FISUB instruction, 3-394
FISUBR instruction, 3-397
FLD instruction, 3-352
FLD1 instruction, 3-354
FLDCW instruction, 3-356
FLDENV instruction, 3-358
FLDL2E instruction, 3-354
FLDL2T instruction, 3-354
FLDLG2 instruction, 3-354
FLDLN2 instruction, 3-354
FLDPI instruction, 3-354
FLDZ instruction, 3-354
Floating point instructions
  machine encodings, B-61
Floating-point exceptions
  SSE and SSE2 SIMD, 3-16, 3-17
  x87 FPU, 3-16
Flushing
  caches, 3-483, 6-2
  TLB entry, 3-485
FMUL instruction, 3-360
FMULP instruction, 3-360
FNCLEX instruction, 3-320
FNINIT instruction, 3-345
FNOP instruction, 3-363
FNSAVE instruction, 3-375
FNSTCW instruction, 3-388
FNSTENV instruction, 3-358, 3-390
FNSTSW instruction, 3-392
FPATAN instruction, 3-364
FPREM instruction, 3-366
FPREM1 instruction, 3-368
FPTAN instruction, 3-370
FRNDINT instruction, 3-372
FRSTOR instruction, 3-373
FS register, 3-540
FSAVE instruction, 3-375
FSAVE/FNSAVE instructions, 3-373
FSCALE instruction, 3-378
FSIN instruction, 3-380
FSINCOS instruction, 3-382
FSQRT instruction, 3-384
FST instruction, 3-386
FSTCW instruction, 3-388
FSTENV instruction, 3-390
FSTP instruction, 3-386
FSTSW instruction, 3-392
FSUB instruction, 3-394
FSUBP instruction, 3-394
FSUBR instruction, 3-397
FSUBRP instruction, 3-397
FTST instruction, 3-400
FUCOM instruction, 3-402
FUCOMI instruction, 3-327
FUCOMIP instruction, 3-327
FUCOMP instruction, 3-402
FUCOMPP instruction, 3-402
FXAM instruction, 3-405
FXCH instruction, 3-407
FXRSTOR instruction, 3-409
FXSAVE instruction, 3-412, 5-776, 6-36, 6-49, 6-54, 6-58, 6-61, 6-64, 6-67, 6-70
FXTRACT instruction, 3-378, 3-420
FYL2X instruction, 3-422
FYL2XP1 instruction, 3-424

G

GDT (global descriptor table), 3-553
GDTR (global descriptor table register), 3-553, 4-629
General-purpose instructions
  64-bit encodings, B-18
  non-64-bit encodings, B-7
General-purpose registers
  moving value to and from, 4-29
  popping all, 4-403
  pushing all, 4-526

GETSEC, 7-1, 7-2, 7-5
GS register, 3-540

H

HADDPD instruction, 3-433, 3-434
HADDPS instruction, 3-436
HLT instruction, 3-439
HSUBPD instruction, 3-442
HSUBPS instruction, 3-445

I

IA-32e mode
  introduction, 2-7, 2-12, 2-21, 2-36
  see 64-bit mode
  see compatibility mode
IDIV instruction, 3-448
IDT (interrupt descriptor table), 3-468, 3-553
IDTR (interrupt descriptor table register), 3-553, 4-655
IF (interrupt enable) flag, EFLAGS register, 3-148, 4-674
Immediate operands, 2-3
IMUL instruction, 3-451
IN instruction, 3-455
INC instruction, 3-457, 3-565
Index (operand addressing), 2-3
Initialization x87 FPU, 3-345
initiating logical processor, 7-4, 7-5, 7-10, 7-22, 7-23
INS instruction, 3-464, 4-563
INSB instruction, 3-464
INSD instruction, 3-464
INSERTPS- Insert Scalar Single Precision Floating-Point Value, 3-461
instruction encodings, B-58, B-64, B-70
Instruction format
  base field, 2-3
  description of reference information, 3-1
  displacement, 2-3
  immediate, 2-3
  index field, 2-3
  Mod field, 2-3
  ModR/M byte, 2-3
  opcode, 2-2
  prefixes, 2-1
  reg/opcode field, 2-3
  r/m field, 2-3
  scale field, 2-3
  SIB byte, 2-3
  See also: machine instructions, opcodes
Instruction reference, nomenclature, 3-1
Instruction set, reference, 3-1
INSW instruction, 3-464
INT 3 instruction, 3-467
Integer, storing, x87 FPU data type, 3-347
Intel 64 architecture
  instruction format, 2-1
Intel® Trusted Execution Technology, 7-3
Inter-privilege level
  call, CALL instruction, 3-121
  return, RET instruction, 4-569
Interrupts
  returning from, 3-298, 3-301, 3-490, 3-556
  software, 3-467
INTn instruction, 3-467
INTO instruction, 3-467
Intrinsics
  compiler functional equivalents, C-1
  composite, C-14
  description of, 3-12
  list of, C-1
  simple, C-2
INVD instruction, 3-483
INVLPG instruction, 3-485
IOPL (I/O privilege level) field, EFLAGS register, 3-148
IRET instruction, 3-298, 3-301, 3-490, 3-556
IRETD instruction, 3-298, 3-301, 3-490, 3-556

J

Jcc instructions, 3-499
JMP instruction, 3-504
Jump operation, 3-504

L

LAHF instruction, 3-532
LAR instruction, 3-533
Last branch
  interrupt & exception recording
    description of, 4-584
LDDQU instruction, 3-537
LDMXCSR instruction, 3-539, 4-542, 6-6
LDS instruction, 3-540
LDT (local descriptor table), 3-558
LDTR (local descriptor table register), 3-558, 4-657
LEA instruction, 3-547
LEAVE instruction, 3-550
LES instruction, 3-540
LFENCE instruction, 3-552
LFS instruction, 3-540
LGDT instruction, 3-553
LGS instruction, 3-540
LIDT instruction, 3-553
LLDT instruction, 3-558, 4-608
LMSW instruction, 3-560
Load effective address operation, 3-547
LOADIWKEY—Load Internal Wrapping Key, 3-562
LOCK prefix, 3-10, 3-14, 3-60, 3-114, 3-116, 3-118, 3-194, 3-262, 3-457, 3-565, 4-158, 4-161, 4-163, 4-614, 4-686, 6-27, 6-32, 6-40
Locking operation, 3-565
LODS instruction, 3-567, 4-563
LODSB instruction, 3-567
LODSD instruction, 3-567
LODSQ instruction, 3-567
LODSW instruction, 3-567
Log epsilon, x87 FPU operation, 3-422
Log (base 2), x87 FPU operation, 3-424
LOOP instructions, 3-570
LOOPcc instructions, 3-570
LSL instruction, 3-573
LSS instruction, 3-540
LTR instruction, 3-576
LZCNT - Count the Number of Leading Zero Bits, 3-578

M

Machine instructions
  64-bit mode, B-1
  condition test (tttn) field, B-6
  direction bit (d) field, B-6

floating-point instruction encodings, B-61
general description, B-1
general-purpose encodings, B-7—B-37
legacy prefixes, B-1
MMX encodings, B-38—B-41
opcode fields, B-2
operand size (w) bit, B-4
P6 family encodings, B-41
Pentium processor family encodings, B-37
reg (reg) field, B-3, B-4
REX prefixes, B-2
segment register (sreg) field, B-5
sign-extend (s) bit, B-5
SIMD 64-bit encodings, B-37
special 64-bit encodings, B-61
special fields, B-2
special-purpose register (eee) field, B-5
SSE encodings, B-42—B-47
SSE2 encodings, ??—B-56
SSE2 encodingsSSE2 extensions
instruction encodings, B-47—??
SSE3 encodings, B-57—B-58
SSSE3 encodings, B-58—B-60
VMX encodings, B-112—??, B-113
See also: opcodes
Machine status word, CR0 register, 3-560, 4-659
MASKMOVDQU instruction, 4-35
MAXPD- Maximum of Packed Double Precision Floating-Point Values, 4-5
MAXPS- Maximum of Packed Single Precision Floating-Point Values, 4-8
MAXSD- Return Maximum Scalar Double Precision Floating-Point Value, 4-11
MAXSS- Return Maximum Scalar Single Precision Floating-Point Value, 4-13
measured environment, 7-1
Measured Launched Environment, 7-1, 7-26
MFENCE instruction, 4-15
MINPD- Minimum of Packed Double Precision Floating-Point Values , 4-16
MINPS- Minimum of Packed Single Precision Floating-Point Values , 4-19
MINSD- Return Minimum Scalar Double Precision Floating-Point Value, 4-22
MINSS- Return Minimum Scalar Single Precision Floating-Point Value, 4-24
MLE, 7-1
MMX instructions
encodings, B-38
Mod field, instruction format, 2-3
ModR/M byte, 2-3
16-bit addressing forms, 2-5
32-bit addressing forms of, 2-6
description of, 2-3
MONITOR instruction, 4-26
MOV instruction, 4-28
MOV instruction (control registers), 4-32, 4-50, 4-52
MOV instruction (debug registers), 4-35, 4-45
MOVAPD- Move Aligned Packed Double Precision Floating-Point Values, 4-37
MOVAPS- Move Aligned Packed Single Precision Floating-Point Values, 4-41
MOVD instruction, 4-45
MOVDDUP- Replicate Double FP Values, 4-47
MOVDQ2Q instruction, 4-58
MOVDQA- Move Aligned Packed Integer Values, 4-59
MOVDQU- Move Unaligned Packed Integer Values, 4-64
MOVHLPS - Move Packed Single Precision Floating-Point Values High to Low, 4-72
MOVHPD- Move High Packed Double Precision Floating-Point Values, 4-74
MOVHPS- Move High Packed Single Precision Floating-Point Values, 4-76
MOVLPD- Move Low Packed Double Precision Floating-Point Values, 4-80
MOVLPS- Move Low Packed Single Precision Floating-Point Values , 4-82
MOVMSKPD instruction, 4-84
MOVMSKPS instruction, 4-86
MOVNTDQ instruction, 4-103
MOVNTDQ- Store Packed Integers Using Non-Temporal Hint, 4-88
MOVNTI instruction, 4-103
MOVNTPD- Store Packed Double Precision Floating-Point Values Using Non-Temporal Hint, 4-94
MOVNTPS- Store Packed Single Precision Floating-Point Values Using Non-Temporal Hint, 4-96
MOVNTQ instruction, 4-98
MOVQ instruction, 4-45, 4-99
MOVQ2DQ instruction, 4-102
MOVS instruction, 4-104, 4-563
MOVSB instruction, 4-104
MOVSD instruction, 4-104
MOVSD- Move or Merge Scalar Double Precision Floating-Point Value, 4-108
MOVSHDUP- Replicate Single FP Values, 4-111
MOVSLDUP- Replicate Single FP Values, 4-114
MOVSQ instruction, 4-104
MOVSS- Move or Merge Scalar Single Precision Floating-Point Value, 4-117
MOVSW instruction, 4-104
MOVSX instruction, 4-120
MOVSXD instruction, 4-120
MOVUPD- Move Unaligned Packed Double Precision Floating-Point Values, 4-122
MOVUPS- Move Unaligned Packed Single Precision Floating-Point Values, 4-126
MOVZX instruction, 4-130
MSRs (model specific registers)
reading, 4-544
MUL instruction, 3-5, 4-140
MULPD- Multiply Packed Double Precision Floating-Point Values, 4-142
MULPS- Multiply Packed Single Precision Floating-Point Values, 4-145
MULSD- Multiply Scalar Double Precision Floating-Point Values, 4-148
MULSS- Multiply Scalar Single Precision Floating-Point Values, 4-150
Multi-byte no operation, 4-158, 4-160, B-13
MULX - Unsigned Multiply Without Affecting Flags, 4-152
MVMM, 7-1, 7-5, 7-38
MWAIT instruction, 4-154

N

NaN. testing for, 3-400
Near
return, RET instruction, 4-569
NEG instruction, 3-565, 4-158
NetBurst microarchitecture (see Intel NetBurst microarchitecture)

No operation, 4-158, 4-160, B-12
Nomenclature, used in instruction reference pages, 3-1
NOP instruction, 4-160
NOT instruction, 3-565, 4-161
NT (nested task) flag, EFLAGS register, 3-490

O

OF (carry) flag, EFLAGS register, 3-452
OF (overflow) flag, EFLAGS register, 3-14, 3-467, 4-140, 4-614, 4-640, 4-643, 4-685
Opcode format, 2-2
Opcodes
  addressing method codes for, A-1
  extensions, A-17
  extensions tables, A-18
  group numbers, A-17
  integers
    one-byte opcodes, A-7
    two-byte opcodes, A-7
  key to abbreviations, A-1
  look-up examples, A-3, A-17, A-21
  ModR/M byte, A-17
  one-byte opcodes, A-3, A-7
  opcode maps, A-1
  operand type codes for, A-2
  register codes for, A-3
  superscripts in tables, A-6
  two-byte opcodes, A-4, A-5, A-7
  VMX instructions, B-112, B-113
  x87 ESC instruction opcodes, A-21
OR instruction, 3-565, 4-163
ORPD, 4-165
ORPS- Bitwise Logical OR of Packed Single Precision Floating-Point Values, 4-168
OUT instruction, 4-171
OUTS instruction, 4-173, 4-563
OUTSB instruction, 4-173
OUTSD instruction, 4-173
OUTSW instruction, 4-173
Overflow exception (#OF), 3-467

P

P6 family processors
  machine encodings, B-41
PABSB instruction, 4-177, 4-191, 5-154, 5-574, 5-585, 5-600
PABSD instruction, 4-177, 4-191, 5-154, 5-574, 5-585, 5-600
PABSW instruction, 4-177, 4-191, 5-154, 5-574, 5-585, 5-600
PACKSSDW instruction, 4-183
PACKSSWB instruction, 4-183
PACKUSWB instruction, 4-196
PADDB/PADDW/PADDD/PADDQ - Add Packed Integers, 4-201
PADDSB instruction, 4-208
PADDSW instruction, 4-208
PADDUSB instruction, 4-212
PADDUSW instruction, 4-212
PALIGNR instruction, 4-216
PAND instruction, 4-220
PANDN instruction, 4-223
GETSEC, 7-4
PAUSE instruction, 4-226
PAVGB instruction, 4-227
PAVGW instruction, 4-227
PCE flag, CR4 register, 4-551
PCMPEQB instruction, 4-245
PCMPEQD instruction, 4-245
PCMPEQW instruction, 4-245
PCMPGTB instruction, 4-258
PCMPGTD instruction, 4-258
PCMPGTW instruction, 4-258
PDEP - Parallel Bits Deposit, 4-282
PE (protection enable) flag, CR0 register, 3-560
Pentium processor family processors
  machine encodings, B-37
PEXT - Parallel Bits Extract, 4-284
PEXTRW instruction, 4-289
PHADDD instruction, 4-294
PHADDSW instruction, 4-292
PHADDW instruction, 4-294
PH—Fused Multiply-Add of Packed FP16 Values, 5-214
PH—Fused Multiply-Subtract of Packed FP16 Values, 5-263
PHSUBD instruction, 4-302
PHSUBSW instruction, 4-300
PHSUBW instruction, 4-302
Pi, 3-354
PINSRW instruction, 4-308, 4-437
PMADDUBSW instruction, 4-310
PMADDUDSW instruction, 4-310
PMADDWD instruction, 4-313
PMULHRSW instruction, 4-375
PMULHUW instruction, 4-379
PMULHW instruction, 4-383
PMULLW instruction, 4-391
PMULUDQ instruction, 4-395
POP instruction, 4-398
POPA instruction, 4-403
POPAD instruction, 4-403
POPF instruction, 4-407
POPFD instruction, 4-407
POPFQ instruction, 4-407
POR instruction, 4-411
PREFETCHh instruction, 4-414
PREFETCHWT1—Prefetch Vector Data Into Caches with Intent to Write and T1 Hint, 4-418
Prefixes
  Address-size override prefix, 2-2
  Branch hints, 2-2
  branch hints, 2-2
  instruction, description of, 2-1
  legacy prefix encodings, B-1
  LOCK, 2-1, 3-565
  Operand-size override prefix, 2-2
  REP or REPE/REPZ, 2-1
  REPNE/REPNZ, 2-1
  REP/REPE/REPZ/REPNE/REPNZ, 4-563, 4-565, 4-567
  REX prefix encodings, B-2
  Segment override prefixes, 2-2
PSADBW instruction, 4-418
PSHUFB instruction, 4-422
PSHUFD instruction, 4-426
PSHUFHW instruction, 4-430
PSHUFLW instruction, 4-433
PSHUFW instruction, 4-436
PSIGNB instruction, 4-437
PSIGND instruction, 4-437
PSIGNW instruction, 4-437
PSLLD instruction, 4-443
PSLLDQ instruction, 4-441
PSLLQ instruction, 4-443

PSLLW instruction, 4-443
PSRAD instruction, 4-455
PSRAW instruction, 4-455
PSRLD instruction, 4-467
PSRLDQ instruction, 4-465
PSRLQ instruction, 4-467
PSRLW instruction, 4-467
PSUBB instruction, 4-479
PSUBD instruction, 4-479
PSUBQ instruction, 4-487
PSUBSB instruction, 4-490
PSUBSW instruction, 4-490
PSUBUSB instruction, 4-494
PSUBUSW instruction, 4-494
PSUBW instruction, 4-479
PTEST- Packed Bit Test, 3-527
PUNPCKHBW instruction, 4-502
PUNPCKHDQ instruction, 4-502
PUNPCKHQDQ instruction, 4-502
PUNPCKHWD instruction, 4-502
PUNPCKLBW instruction, 4-512
PUNPCKLDQ instruction, 4-512
PUNPCKLQDQ instruction, 4-512
PUNPCKLWD instruction, 4-512
PUSH instruction, 4-522
PUSHA instruction, 4-526
PUSHAD instruction, 4-526
PUSHF instruction, 4-528
PUSHFD instruction, 4-528
PXOR instruction, 4-530

R

RC (rounding control) field, x87 FPU control word, 3-347, 3-354, 3-386
RCL instruction, 4-533
RCPPS instruction, 4-538
RCPSS instruction, 4-540
RCR instruction, 4-533
RDMSR instruction, 4-544, 4-546, 4-559
RDPMC instruction, 4-549, 4-551, 6-15
RDTSC instruction, 4-554, 4-559, 4-561
Reg/opcode field, instruction format, 2-3
Remainder, x87 FPU operation, 3-368
REP/REPE/REPZ/REPNE/REPNZ prefixes, 3-182, 3-465, 4-173, 4-563, 4-565, 4-567
Responding logical processor, 7-4
responding logical processor, 7-4, 7-5
RET instruction, 4-569
REX prefixes
  addressing modes, 2-9
  and INC/DEC, 2-8
  encodings, 2-8, B-2
  field names, 2-9
  ModR/M byte, 2-8
  overview, 2-7
  REX.B, 2-8
  REX.R, 2-8
  REX.W, 2-8
  special encodings, 2-10
RIP-relative addressing, 2-11
ROL instruction, 4-533
ROR instruction, 4-533
RORX - Rotate Right Logical Without Affecting Flags, 4-582
Rounding
  modes, floating-point operations, 4-584
Rounding control (RC) field
  MXCSR register, 4-584
  x87 FPU control word, 4-584
Rounding, round to integer, x87 FPU operation, 3-372
RPL field, 3-75
RSM instruction, 4-592
RSQRTPS instruction, 4-594
RSQRTSS instruction, 4-596
R/m field, instruction format, 2-3

S

Safer Mode Extensions, 7-1
SAHF instruction, 4-601
SAL instruction, 4-603
SAR instruction, 4-603
SBB instruction, 3-565, 4-613
Scale (operand addressing), 2-3
Scale, x87 FPU operation, 3-378
Scan string instructions, 4-616
SCAS instruction, 4-567, 4-616
SCASB instruction, 4-616
SCASD instruction, 4-616
SCASW instruction, 4-616
Segment
  descriptor, segment limit, 3-573
  limit, 3-573
  registers, moving values to and from, 4-29
  selector, RPL field, 3-75
GETSEC, 7-2, 7-4, 7-5
SENTER sleep state, 7-10
SETcc instructions, 4-623
GETSEC, 7-4
SF (sign) flag, EFLAGS register, 3-14
SFENCE instruction, 4-628
SGDT instruction, 4-629
SHAF instruction, 4-601
SH—Fused Multiply-Add of Scalar FP16 Values, 5-229
SH—Fused Multiply-Subtract of Scalar FP16 Values, 5-279
Shift instructions, 4-603
SHL instruction, 4-603
SHLD instruction, 4-640
SHR instruction, 4-603
SHRD instruction, 4-643
SHUFPD - Shuffle Packed Double Precision Floating-Point Values, 4-646, 4-687
SHUFPS - Shuffle Packed Single Precision Floating-Point Values, 4-651
SIB byte, 2-3
  32-bit addressing forms of, 2-7, 2-21
  description of, 2-3
SIDT instruction, 4-629, 4-655
Significand, extracting from floating-point number, 3-420
SIMD floating-point exceptions, unmasking, effects of, 3-539, 4-542, 6-6
Sine, x87 FPU operation, 3-380, 3-382
SINIT, 7-4
SLDT instruction, 4-657
GETSEC, 7-4
SMSW instruction, 4-659
SQRTPD—Square Root of Double Precision Floating-Point Values, 4-661
SQRTPS- Square Root of Single Precision Floating-Point Values, 4-664

SQRTSD - Compute Square Root of Scalar Double Precision Floating-Point Value, 4-667
SQRTSS - Compute Square Root of Scalar Single Precision Floating-Point Value, 4-669
Square root, Fx87 PU operation, 3-384
SS register, 3-540, 4-29, 4-399
SSE extensions
  cacheability instruction encodings, B-47
  floating-point encodings, B-42
  instruction encodings, B-42
  integer instruction encodings, B-46
  memory ordering encodings, B-47
SSE2 extensions
  cacheability instruction encodings, B-56
  floating-point encodings, B-48
  integer instruction encodings, B-52
SSE3 extensions
  event mgmt instruction encodings, B-57
  floating-point instruction encodings, B-57
  integer instruction encodings, B-57, B-58
SSSE3 extensions, B-58, B-64, B-70
Stack, pushing values on, 4-522
Status flags, EFLAGS register, 3-159, 3-161, 3-322, 3-327, 3-502, 4-624, 4-721
STC instruction, 4-672
STD instruction, 4-673
STI instruction, 4-674
STMXCSR instruction, 4-676
STOS instruction, 4-563, 4-677
STOSB instruction, 4-677
STOSD instruction, 4-677
STOSQ instruction, 4-677
STOSW instruction, 4-677
STR instruction, 4-680
String instructions, 3-181, 3-464, 3-567, 4-104, 4-173, 4-616, 4-677
SUB instruction, 3-7, 3-260, 3-565, 4-685
SUBPD- Subtract Packed Double Precision Floating-Point Values, 4-687
SUBPS- Subtract Packed Single Precision Floating-Point Values, 4-690
SUBSD- Subtract Scalar Double Precision Floating-Point Values, 4-693
SUBSS- Subtract Scalar Single Precision Floating-Point Values, 4-695
SWAPGS instruction, 4-697
SYSCALL instruction, 4-699
SYSENTER instruction, 4-702
SYSEXIT instruction, 4-706
SYSRET instruction, 4-709

T

Tangent, x87 FPU operation, 3-370
Task register
  storing, 4-680
Task switch
  CALL instruction, 3-121
  return from nested task, IRET instruction, 3-298, 3-301, 3-490, 3-556
TEST instruction, 4-721, 5-771
Time-stamp counter, reading, 4-559, 4-561
TLB entry, invalidating (flushing), 3-485
Trusted Platform Module, 7-5
TS (task switched) flag, CR0 register, 3-152
TSS, relationship to task register, 4-680
TZCNT - Count the Number of Trailing Zero Bits, 4-731

U

UCOMISD - Unordered Compare Scalar Double Precision Floating-Point Values and Set EFLAGS, 4-733
UCOMISD instruction, 4-731
UCOMISS - Unordered Compare Scalar Single Precision Floating-Point Values and Set EFLAGS, 4-735
UD2 instruction, 4-737
Undefined, format opcodes, 3-400
Unordered values, 3-324, 3-400, 3-402
UNPCKHPD- Unpack and Interleave High Packed Double Precision Floating-Point Values, 4-744
UNPCKHPS- Unpack and Interleave High Packed Single Precision Floating-Point Values, 4-748
UNPCKLPD- Unpack and Interleave Low Packed Double Precision Floating-Point Values, 4-752
UNPCKLPS- Unpack and Interleave Low Packed Single Precision Floating-Point Values, 4-756
URDMSR, 4-760
UWRMSR, 4-762

V

VADDPH—Add Packed FP16 Values, 5-1
VADDSH—Add Scalar FP16 Values, 5-3
VALIGND/VALIGNQ—Align Doubleword/Quadword Vectors, 5-4
VBLENDMPD/VBLENDMPS—Blend Float64/Float32 Vectors Using an OpMask Control, 5-9
VBROADCAST—Load with Broadcast Floating-Point Data, 5-12
VCMPPH—Compare Packed FP16 Values, 5-20
VCMPSH—Compare Scalar FP16 Values, 5-22
VCOMISH—Compare Scalar Ordered FP16 Values and Set EFLAGS, 5-24
VCOMPRESSPD—Store Sparse Packed Double Precision Floating-Point Values Into Dense Memory, 5-26
VCOMPRESSPS—Store Sparse Packed Single Precision Floating-Point Values Into Dense Memory, 5-28
VCVTDQ2PH—Convert Packed Signed Doubleword Integers to Packed FP16 Values, 5-30
VCVTNE2PS2BF16—Convert Two Packed Single Data to One Packed BF16 Data, 5-32
VCVTNEPS2BF16—Convert Packed Single Data to Packed BF16 Data, 5-38
VCVTPD2PH—Convert Packed Double Precision FP Values to Packed FP16 Values, 5-40
VCVTPD2QQ—Convert Packed Double Precision Floating-Point Values to Packed Quadword Integers, 5-42
VCVTPD2UDQ—Convert Packed Double Precision Floating-Point Values to Packed Unsigned Doubleword Integers, 5-44
VCVTPD2UQQ—Convert Packed Double Precision Floating-Point Values to Packed Unsigned Quadword Integers, 5-47
VCVTPH2DQ—Convert Packed FP16 Values to Signed Doubleword Integers, 5-50
VCVTPH2PD—Convert Packed FP16 Values to FP64 Values, 5-52
VCVTPH2QQ—Convert Packed FP16 Values to Signed Quadword Integer Values, 5-58
VCVTPH2UDQ—Convert Packed FP16 Values to Unsigned Doubleword Integers, 5-60
VCVTPH2UQQ—Convert Packed FP16 Values to Unsigned Quadword Integers, 5-62
VCVTPH2UW—Convert Packed FP16 Values to Unsigned Word Integers, 5-64

VCVTPH2W—Convert Packed FP16 Values to Signed Word Integers, 5-66
VCVTPS2PH—Convert Single Precision FP Value to 16-bit FP Value, 5-68
VCVTPS2PHX—Convert Packed Single Precision Floating-Point Values to Packed FP16 Values, 5-72
VCVTPS2UDQ—Convert Packed Single Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values, 5-76
VCVTPS2UQQ—Convert Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values, 5-79
VCVTQQ2PD—Convert Packed Quadword Integers to Packed Double Precision Floating-Point Values, 5-81
VCVTQQ2PH—Convert Packed Signed Quadword Integers to Packed FP16 Values, 5-83
VCVTQQ2PS—Convert Packed Quadword Integers to Packed Single Precision Floating-Point Values, 5-85
VCVTSD2SH—Convert Low FP64 Value to an FP16 Value, 5-87
VCVTSD2USI—Convert Scalar Double Precision Floating-Point Value to Unsigned Doubleword Integer, 5-88
VCVTSH2SD—Convert Low FP16 Value to an FP64 Value, 5-90
VCVTSH2SI—Convert Low FP16 Value to Signed Integer, 5-91
VCVTSH2SS—Convert Low FP16 Value to FP32 Value, 5-93
VCVTSH2USI—Convert Low FP16 Value to Unsigned Integer, 5-94
VCVTSI2SH—Convert a Signed Doubleword/Quadword Integer to an FP16 Value, 5-96
VCVTSS2SH—Convert Low FP32 Value to an FP16 Value, 5-98
VCVTSS2USI—Convert Scalar Single Precision Floating-Point Value to Unsigned Doubleword Integer, 5-99
VCVTTPD2QQ—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Quadword Integers, 5-101
VCVTTPD2UDQ—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Unsigned Doubleword Integers, 5-103
VCVTTPD2UQQ—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Unsigned Quadword Integers, 5-105
VCVTTPH2DQ—Convert with Truncation Packed FP16 Values to Signed Doubleword Integers, 5-107
VCVTTPH2QQ—Convert with Truncation Packed FP16 Values to Signed Quadword Integers, 5-109
VCVTTPH2UDQ—Convert with Truncation Packed FP16 Values to Unsigned Doubleword Integers, 5-111
VCVTTPH2UQQ—Convert with Truncation Packed FP16 Values to Unsigned Quadword Integers, 5-113
VCVTTPH2UW—Convert Packed FP16 Values to Unsigned Word Integers, 5-115
VCVTTPH2W—Convert Packed FP16 Values to Signed Word Integers, 5-117
VCVTTPS2QQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Signed Quadword Integer Values, 5-123
VCVTTPS2UDQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values, 5-121
VCVTTPS2UQQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values, 5-123
VCVTTSD2USI—Convert With Truncation Scalar Double Precision Floating-Point Value to Unsigned Integer, 5-125
VCVTTSH2SI—Convert with Truncation Low FP16 Value to a Signed Integer, 5-127
VCVTTSH2USI—Convert with Truncation Low FP16 Value to an Unsigned Integer, 5-128
VCVTTSS2USI—Convert With Truncation Scalar Single Precision Floating-Point Value to Unsigned Integer, 5-129
VCVTUDQ2PD—Convert Packed Unsigned Doubleword Integers to Packed Double Precision Floating-Point Values, 5-131
VCVTUDQ2PH—Convert Packed Unsigned Doubleword Integers to Packed FP16 Values, 5-133
VCVTUDQ2PS—Convert Packed Unsigned Doubleword Integers to Packed Single Precision Floating-Point Values, 5-135
VCVTUQQ2PD—Convert Packed Unsigned Quadword Integers to Packed Double Precision Floating-Point Values, 5-138
VCVTUQQ2PH—Convert Packed Unsigned Quadword Integers to Packed FP16 Values, 5-140
VCVTUQQ2PS—Convert Packed Unsigned Quadword Integers to Packed Single Precision Floating-Point Values, 5-142
VCVTUSI2SD—Convert Unsigned Integer to Scalar Double Precision Floating-Point Value, 5-144
VCVTUSI2SH—Convert Unsigned Doubleword Integer to an FP16 Value, 5-146
VCVTUSI2SS—Convert Unsigned Integer to Scalar Single Precision Floating-Point Value, 5-148
VCVTUW2PH—Convert Packed Unsigned Word Integers to FP16 Values, 5-150
VCVTW2PH—Convert Packed Signed Word Integers to FP16 Values, 5-152
VDBPSADBW—Double Block Packed Sum-Absolute-Differences (SAD) on Unsigned Bytes, 5-154
VDIVPH—Divide Packed FP16 Values, 5-157
VDIVSH—Divide Scalar FP16 Values, 5-159
VDPBF16PS—Dot Product of BF16 Pairs Accumulated Into Packed Single Precision, 5-160
VERR instruction, 5-162
VERR/VERW—Verify a Segment for Reading or Writing, 5-168
Version information, processor, 3-203
VERW instruction, 5-162
VEX, 3-3
VEXPANDPD—Load Sparse Packed Double Precision Floating-Point Values From Dense Memory, 5-164
VEXPANDPS—Load Sparse Packed Single Precision Floating-Point Values From Dense Memory, 5-166
VEXTRACTF128/VEXTRACTF32x4/VEXTRACTF64x2/VEXTRACTF32x8/VEXTRACTF64x4— Extract Packed Floating-Point Values, 5-168
VEXTRACTI128/VEXTRACTI32x4/VEXTRACTI64x2/VEXTRACTI32x8/VEXTRACTI64x4—Extract Packed Integer Values, 5-174
VEX.B, 3-3
VEX.L, 3-3, 3-4
VEX.mmmmm, 3-3
VEX.pp, 3-4
VEX.R, 3-4
VEX.W, 3-3
VEX.X, 3-3
VFCMADDCPH/VFMADDCPH—Complex Multiply and Accumulate FP16 Values, 5-180
VFCMADDCSH/VFMADDCSH—Complex Multiply and Accumulate Scalar FP16 Values, 5-183
VFCMULCPH/VFMULCPH—Complex Multiply FP16 Values, 5-185
VFCMULCSH/VFMULCSH—Complex Multiply Scalar FP16 Values, 5-189
VFIXUPIMMPD—Fix Up Special Packed Float64 Values, 5-191
VFIXUPIMMPS—Fix Up Special Packed Float32 Values, 5-195
VFIXUPIMMSD—Fix Up Special Scalar Float64 Value, 5-199
VFIXUPIMMSS—Fix Up Special Scalar Float32 Value, 5-203

VFMADD132PD/VFMADD213PD/VFMADD231PD—Fused Multiply-Add of Packed Double Precision Floating-Point Values, 5-207
VFMADD132PS/VFMADD213PS/VFMADD231PS—Fused Multiply-Add of Packed Single Precision Floating-Point Values, 5-220
VFMADD132SD/VFMADD213SD/VFMADD231SD—Fused Multiply-Add of Scalar Double Precision Floating-Point Values, 5-226
VFMADD132SS/VFMADD213SS/VFMADD231SS—Fused Multiply-Add of Scalar Single Precision Floating-Point Values, 5-232
VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD—Fused Multiply-Alternating Add/Subtract of Packed Double Precision Floating-Point Values, 5-235
VFMADDSUB132PH/VFMADDSUB213PH/VFMADDSUB231PH—Fused Multiply-Alternating Add/Subtract of Packed FP16 Values, 5-243
VFMADDSUB132PS/VFMADDSUB213PS/VFMADDSUB231PS—Fused Multiply-Alternating Add/Subtract of Packed Single Precision Floating-Point Values, 5-248
VFMSUB132PS/VFMSUB213PS/VFMSUB231PS— Fused Multiply-Subtract of Packed Single Precision Floating-Point Values, 5-269
VFMSUB132SD/VFMSUB213SD/VFMSUB231SD— Fused Multiply-Subtract of Scalar Double Precision Floating-Point Values, 5-276
VFMSUB132SS/VFMSUB213SS/VFMSUB231SS—Fused Multiply-Subtract of Scalar Single Precision Floating-Point Values, 5-282
VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD—Fused Multiply-Alternating Subtract/Add of Packed Double Precision Floating-Point Values, 5-285
VFMSUBADD132PH/VFMSUBADD213PH/VFMSUBADD231PH—Fused Multiply-Alternating Subtract/Add of Packed FP16 Values, 5-292
VFMSUBADD132PS/VFMSUBADD213PS/VFMSUBADD231PS—Fused Multiply-Alternating Subtract/Add of Packed Single Precision Floating-Point Values, 5-297
VFNMADD132PD/VFMADD213PD/VFMADD231PD —Fused Negative Multiply-Add of Packed Double Precision Floating-Point Values, 5-305
VFNMADD132PS/VFNMADD213PS/VFNMADD231PS—Fused Negative Multiply-Add of Packed Single Precision Floating-Point Values, 5-312
VFNMADD132SD/VFNMADD213SD/VFNMADD231SD—Fused Negative Multiply-Add of Scalar Double Precision Floating-Point Values, 5-319
VFNMADD132SS/VFNMADD213SS/VFNMADD231SS—Fused Negative Multiply-Add of Scalar Single Precision Floating-Point Values, 5-322
VFNMSUB132PD/VFNMSUB213PD/VFNMSUB231PD—Fused Negative Multiply-Subtract of Packed Double Precision Floating-Point Values, 5-325
VFNMSUB132PS/VFNMSUB213PS/VFNMSUB231PS—Fused Negative Multiply-Subtract of Packed Single Precision Floating-Point Values, 5-332
VFNMSUB132SD/VFNMSUB213SD/VFNMSUB231SD—Fused Negative Multiply-Subtract of Scalar Double Precision Floating-Point Values, 5-339
VFNMSUB132SS/VFNMSUB213SS/VFNMSUB231SS—Fused Negative Multiply-Subtract of Scalar Single Precision Floating-Point Values, 5-342
VFPCLASSPD—Tests Types of Packed Float64 Values, 5-345
VFPCLASSPH—Test Types of Packed FP16 Values, 5-348
VFPCLASSPS—Tests Types of Packed Float32 Values, 5-351
VFPCLASSSD—Tests Type of a Scalar Float64 Value, 5-353
VFPCLASSSH—Test Types of Scalar FP16 Values, 5-355
VFPCLASSSS—Tests Type of a Scalar Float32 Value, 5-356
VGATHERDPD/VGATHERQPD—Gather Packed Double Precision Floating-Point Values Using Signed Dword/Qword Indices, 5-358
VGATHERDPS/VGATHERDPD—Gather Packed Single, Packed Double with Signed Dword, 5-362
VGATHERDPS/VGATHERQPS—Gather Packed SP FP values Using Signed Dword/Qword Indices, 5-365
VGATHERQPS/VGATHERQPD—Gather Packed Single, Packed Double with Signed Qword Indices, 5-369
VGETEXPPD—Convert Exponents of Packed Double Precision Floating-Point Values to Double Precision Floating-Point Values, 5-372
VGETEXPPH—Convert Exponents of Packed FP16 Values to FP16 Values, 5-376
VGETEXPPS—Convert Exponents of Packed Single Precision Floating-Point Values to Single Precision Floating-Point Values, 5-379
VGETEXPSD—Convert Exponents of Scalar Double Precision Floating-Point Value to Double Precision Floating-Point Value, 5-383
VGETEXPSH—Convert Exponents of Scalar FP16 Values to FP16 Values, 5-385
VGETEXPSS—Convert Exponents of Scalar Single Precision Floating-Point Value to Single Precision Floating-Point Value, 5-387
VGETMANTPD—Extract Float64 Vector of Normalized Mantissas From Float64 Vector, 5-389
VGETMANTPH—Extract FP16 Vector of Normalized Mantissas from FP16 Vector, 5-393
VGETMANTPS—Extract Float32 Vector of Normalized Mantissas From Float32 Vector, 5-397
VGETMANTSD—Extract Float64 of Normalized Mantissas From Float64 Scalar, 5-400
VGETMANTSH—Extract FP16 of Normalized Mantissa from FP16 Scalar, 5-402
VGETMANTSS—Extract Float32 Vector of Normalized Mantissa From Float32 Vector, 5-404
VINSERTF128/VINSERTF32x4/VINSERTF64x2/VINSERTF32x8/VINSERTF64x4—Insert Packed Floating-Point Values, 5-406
VINSERTI128/VINSERTI32x4/VINSERTI64x2/VINSERTI32x8/VINSERTI64x4—Insert Packed Integer Values, 5-411
Virtual Machine Monitor, 7-1
VM (virtual 8086 mode) flag, EFLAGS register, 3-490
VMASKMOV—Conditional SIMD Packed Loads and Stores, 5-416
VMAXPH—Return Maximum of Packed FP16 Values, 5-419
VMAXSH—Return Maximum of Scalar FP16 Values, 5-421
VMINPH—Return Minimum of Packed FP16 Values, 5-423
VMINSH—Return Minimum Scalar FP16 Value, 5-425
VMM, 7-1
VMOVSH—Move Scalar FP16 Value, 5-427
VMOVW—Move Word, 5-429
VMULPH—Multiply Packed FP16 Values, 5-430
VMULSH—Multiply Scalar FP16 Values, 5-432
VP2INTERSECTD/VP2INTERSECTQ—Compute Intersection Between DWORDS/QUADWORDS to a Pair of Mask Registers, 5-433
VPBLENDD—Blend Packed Dwords, 5-435
VPBLENDMB/VPBLENDMW—Blend Byte/Word Vectors Using an Opmask Control, 5-437

VPBLENDMD/VPBLENDMQ—Blend Int32/Int64 Vectors Using an OpMask Control, 5-439
VPBROADCASTB/W/D/Q—Load With Broadcast Integer Data From General Purpose Register, 5-451
VPBROADCAST—Load Integer and Broadcast, 5-442
VPBROADCASTM—Broadcast Mask to Vector Register, 5-454
VPCMPB/VPCMPUB—Compare Packed Byte Values Into Mask, 5-456
VPCMPD/VPCMPUD—Compare Packed Integer Values Into Mask, 5-459
VPCMPQ/VPCMPUQ—Compare Packed Integer Values into Mask, 5-462
VPCMPW/VPCMPUW—Compare Packed Word Values Into Mask, 5-465
VPCOMPRESSB/VCOMPRESSW—Store Sparse Packed Byte/Word Integer Values Into Dense Memory/Register, 5-468
VPCOMPRESSD—Store Sparse Packed Doubleword Integer Values Into Dense Memory/Register, 5-471
VPCOMPRESSQ—Store Sparse Packed Quadword Integer Values Into Dense Memory/Registe, 5-473
VPCONFLICTD/Q—Detect Conflicts Within a Vector of Packed Dword/Qword Values Into Dense Memory/ Register, 5-475
VPDPBUSD—Multiply and Add Unsigned and Signed Bytes, 5-481
VPDPBUSDS—Multiply and Add Unsigned and Signed Bytes With Saturation, 5-484
VPDPWSSD—Multiply and Add Signed Word Integers, 5-487
VPDPWSSDS—Multiply and Add Signed Word Integers With Saturation, 5-489
VPERM2F128—Permute Floating-Point Values, 5-494
VPERM2I128—Permute Integer Values, 5-496
VPERMB—Permute Packed Bytes Elements, 5-498
VPERMD/VPERMW—Permute Packed Doubleword/Word Elements , 5-500
VPERMI2B—Full Permute of Bytes From Two Tables Overwriting the Index, 5-503
VPERMI2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting the Index, 5-505
VPERMILPD—Permute In-Lane of Pairs of Double Precision Floating-Point Values, 5-511
VPERMILPS—Permute In-Lane of Quadruples of Single Precision Floating-Point Values, 5-517
VPERMPD—Permute Double Precision Floating-Point Elements, 5-522
VPERMPS—Permute Single Precision Floating-Point Elements, 5-526
VPERMQ—Qwords Element Permutation, 5-529
VPERMT2B—Full Permute of Bytes From Two Tables Overwriting a Table, 5-533
VPERMT2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting One Table, 5-535
VPEXPANDB/VPEXPANDW—Expand Byte/Word Values, 5-541
VPEXPANDD—Load Sparse Packed Doubleword Integer Values From Dense Memory/Register, 5-544
VPEXPANDQ—Load Sparse Packed Quadword Integer Values From Dense Memory/Register, 5-546
VPGATHERDD/VPGATHERDQ—Gather Packed Dword, Packed Qword With Signed Dword Indices, 5-548
VPGATHERDD/VPGATHERQD—Gather Packed Dword Values Using Signed Dword/Qword Indices, 5-551
VPGATHERDQ/VPGATHERQQ—Gather Packed Qword values Using Signed Dword/Qword Indices, 5-555
VPGATHERQD/VPGATHERQQ—Gather Packed Dword, Packed Qword with Signed Qword Indices, 5-559
VPLZCNTD/Q—Count the Number of Leading Zero Bits for Packed Dword, Packed Qword Values, 5-562
VPMADD52HUQ—Packed Multiply of Unsigned 52-Bit Unsigned Integers and Add High 52-Bit Products to 64-Bit Accumulators, 5-565
VPMADD52LUQ—Packed Multiply of Unsigned 52-Bit Integers and Add the Low 52-Bit Products to Qword Accumulators, 5-568
VPMASKMOV—Conditional SIMD Integer Packed Loads and Stores , 5-571
VPMOVB2M/VPMOVW2M/VPMOVD2M/VPMOVQ2M—Convert a Vector Register to a Mask, 5-574
VPMOVDB/VPMOVSDB/VPMOVUSDB - Down Convert DWord to Byte, 5-577
VPMOVDB/VPMOVSDB/VPMOVUSDB—Down Convert DWord to Byte, 5-577
VPMOVDW/VPMOVSDW/VPMOVUSDW - Down Convert DWord to Word, 5-581
VPMOVDW/VPMOVSDW/VPMOVUSDW—Down Convert DWord to Word, 5-581
VPMOVM2B/VPMOVM2W/VPMOVM2D/VPMOVM2Q—Convert a Mask Register to a Vector Register, 5-585
VPMOVQB/VPMOVSQB/VPMOVUSQB - Down Convert QWord to Byte, 5-588
VPMOVQB/VPMOVSQB/VPMOVUSQB—Down Convert QWord to Byte, 5-588
VPMOVQD/VPMOVSQD/VPMOVUSQD - Down Convert QWord to DWord, 5-592
VPMOVQD/VPMOVSQD/VPMOVUSQD—Down Convert QWord to DWord, 5-592
VPMOVQW/VPMOVSQW/VPMOVUSQW - Down Convert QWord to Word, 5-596
VPMOVQW/VPMOVSQW/VPMOVUSQW—Down Convert QWord to Word, 5-596
VPMOVWB/VPMOVSWB/VPMOVUSWB—Down Convert Word to Byte, 5-600
VPMULTISHIFTQB—Select Packed Unaligned Bytes From Quadword Sources, 5-604
VPOPCNT—Return the Count of Number of Bits Set to 1 in BYTE/WORD/DWORD/QWORD, 5-606
VPROLD/VPROLVD/VPROLQ/VPROLVQ—Bit Rotate Left, 5-610
VPRORD/VPRORVD/VPRORQ/VPRORVQ—Bit Rotate Right, 5-614
VPSCATTERDD/VPSCATTERDQ/VPSCATTERQD/VPSCATTERQQ—Scatter Packed Dword, Packed Qword with Signed Dword, Signed Qword Indices, 5-618
VPSHLD—Concatenate and Shift Packed Data Left Logical, 5-622
VPSHLDV—Concatenate and Variable Shift Packed Data Left Logical, 5-625
VPSHRD—Concatenate and Shift Packed Data Right Logical, 5-628
VPSHRDV—Concatenate and Variable Shift Packed Data Right Logical, 5-631
VPSHUFBITQMB—Shuffle Bits From Quadword Elements Using Byte Indexes Into Mask, 5-634
VPSLLVW/VPSLLVD/VPSLLVQ—Variable Bit Shift Left Logical, 5-636
VPSRAVW/VPSRAVD/VPSRAVQ—Variable Bit Shift Right Arithmetic, 5-641
VPSRLVW/VPSRLVD/VPSRLVQ—Variable Bit Shift Right Logical, 5-646
VPTERNLOGD/VPTERNLOGQ—Bitwise Ternary Logic, 5-651
VPTESTMB/VPTESTMW/VPTESTMD/VPTESTMQ—Logical AND and Set Mask, 5-654
VPTESTNMB/W/D/Q—Logical NAND and Set, 5-657
VRANGEPD—Range Restriction Calculation for Packed Pairs of Float64 Values, 5-661

VRANGESD—Range Restriction Calculation From a Pair of Scalar Float64 Values, 5-668
VRANGESS—Range Restriction Calculation From a Pair of Scalar Float32 Values, 5-671
VRCP14PD—Compute Approximate Reciprocals of Packed Float64 Values, 5-674
VRCP14PS—Compute Approximate Reciprocals of Packed Float32 Values, 5-676
VRCP14SD—Compute Approximate Reciprocal of Scalar Float64 Value, 5-678
VRCP14SS—Compute Approximate Reciprocal of Scalar Float32 Value, 5-680
VRCPPH—Compute Reciprocals of Packed FP16 Values, 5-682
VRCPSH—Compute Reciprocal of Scalar FP16 Value, 5-684
VREDUCEPD—Perform Reduction Transformation on Packed Float64 Values, 5-685
VREDUCEPH—Perform Reduction Transformation on Packed FP16 Values, 5-688
VREDUCESD—Perform a Reduction Transformation on a Scalar Float64 Value, 5-693
VREDUCESH—Perform Reduction Transformation on Scalar FP16 Value, 5-695
VREDUCESS—Perform Reduction Transformation on a Scalar Float32 Value, 5-697
VRNDSCALEPD—Round Packed Float64 Values to Include a Given Number of Fraction Bits, 5-699
VRNDSCALEPH—Round Packed FP16 Values to Include a Given Number of Fraction Bits, 5-702
VRNDSCALEPS—Round Packed Float32 Values to Include a Given Number of Fraction Bits, 5-705
VRNDSCALESD—Round Scalar Float64 Value to Include a Given Number of Fraction Bits, 5-708
VRNDSCALESH—Round Scalar FP16 Value to Include a Given Number of Fraction Bits, 5-710
VRNDSCALESS—Round Scalar Float32 Value to Include a Given Number of Fraction Bits, 5-712
VRSQRT14PD—Compute Approximate Reciprocals of Square Roots of Packed Float64 Values, 5-714
VRSQRT14PS—Compute Approximate Reciprocals of Square Roots of Packed Float32 Values, 5-716
VRSQRT14SD—Compute Approximate Reciprocal of Square Root of Scalar Float64 Value, 5-718
VRSQRT14SS—Compute Approximate Reciprocal of Square Root of Scalar Float32 Value, 5-720
VRSQRTPH—Compute Reciprocals of Square Roots of Packed FP16 Values, 5-722
VRSQRTSH—Compute Approximate Reciprocal of Square Root of Scalar FP16 Value, 5-724
VSCALEFPD—Scale Packed Float64 Values With Float64 Values, 5-725
VSCALEFPH—Scale Packed FP16 Values with FP16 Values, 5-728
VSCALEFPS—Scale Packed Float32 Values With Float32 Values, 5-731
VSCALEFSD—Scale Scalar Float64 Values With Float64 Values, 5-734
VSCALEFSH—Scale Scalar FP16 Values with FP16 Values, 5-736
VSCALEFSS—Scale Scalar Float32 Value With Float32 Value, 5-738
VSCATTERDPS/VSCATTERDPD/VSCATTERQPS/VSCATTERQPD—Scatter Packed Single, Packed Double with Signed Dword and Qword Indices, 5-740
VSCATTERPF1DPS/VSCATTERPF1QPS/VSCATTERPF1DPD/VSCATTERPF1QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T1 Hint with Intent to Write, 8-38
VSHUFF32x4/VSHUFF64x2/VSHUFI32x4/VSHUFI64x2—Shuffle Packed Values at 128-Bit Granularity, 5-750
VSQRTPH—Compute Square Root of Packed FP16 Values, 5-765
VSQRTSH—Compute Square Root of Scalar FP16 Value, 5-767
VSUBPH—Subtract Packed FP16 Values, 5-768
VSUBSH—Subtract Scalar FP16 Value, 5-770
VTESTPD/VTESTPS—Packed Bit Test, 5-771
VUCOMISH—Unordered Compare Scalar FP16 Values and Set EFLAGS, 5-774
VZEROALL—Zero XMM, YMM, and ZMM Registers, 5-775, 5-776

W

WAIT/FWAIT instructions, 6-1
GETSEC, 7-4
WBINVD instruction, 6-2, 6-4
Write-back and invalidate caches, 6-2
WRMSR instruction, 4-760, 4-762, 6-8, 6-10, 6-12

X

x87 FPU
  checking for pending x87 FPU exceptions, 6-1
  constants, 3-354
  initialization, 3-345
  instruction opcodes, A-21
x87 FPU control word
  loading, 3-356, 3-358
  RC field, 3-347, 3-354, 3-386
  restoring, 3-373
  saving, 3-375, 3-390
  storing, 3-388
x87 FPU data pointer, 3-358, 3-373, 3-375, 3-390
x87 FPU instruction pointer, 3-358, 3-373, 3-375, 3-390
x87 FPU last opcode, 3-358, 3-373, 3-375, 3-390
x87 FPU status word
  condition code flags, 3-324, 3-340, 3-400, 3-402, 3-405
  loading, 3-358
  restoring, 3-373
  saving, 3-375, 3-390, 3-392
  TOP field, 3-344
  x87 FPU flags affected by instructions, 3-15
x87 FPU tag word, 3-358, 3-373, 3-375, 3-390
XABORT - Transaction Abort, 6-21
XADD instruction, 3-565, 6-27
XCHG instruction, 3-565, 6-32
XCR0, 6-70, 6-71
XEND - Transaction End, 6-34
XGETBV, 6-36, 6-49, 6-54, B-41
XLAB instruction, 6-38
XLAT instruction, 6-38
XOR instruction, 3-565, 6-40
XORPD- Bitwise Logical XOR of Packed Double Precision Floating-Point Values, 6-42
XORPS- Bitwise Logical XOR of Packed Single Precision Floating-Point Values, 5-776, 6-45
XRSTOR, B-41
XSAVE, 6-58, 6-61, 6-64, 6-67, B-41
XSETBV, 6-64, 6-70, B-41
XTEST - Test If In Transactional Execution, 6-73

Z

ZF (zero) flag, EFLAGS register, 3-194, 3-533, 3-570, 3-573, 4-565, 5-162
VF, 5-214, 5-229, 5-263, 5-279